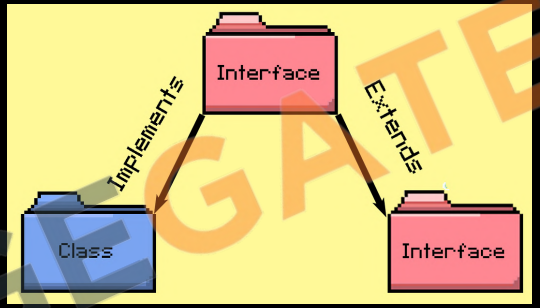
9.3 Interfaces

```java
interface Pet {
    public void play();
}

class Dog extends Animal implements Pet{
    // Dog has its own implementation of run method
    public void run() { }

    public void bark() { }

    // Define the methods of the interface
    public void play() { }
}
```



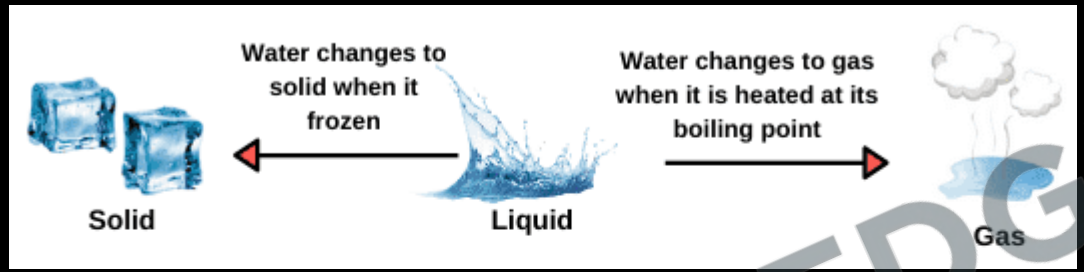
1. Interfaces primarily declare abstract methods for implementation by classes.
2. A class can implement multiple interfaces, allowing for more flexible designs.
3. Interfaces can have default methods with implementation and static methods.
4. Interface methods are inherently public and abstract, except for default and static methods.

CHALLENGE

83. Create an abstract class Shape with an abstract method calculateArea(). Implement two subclasses: Circle and Square. Each subclass should have relevant attributes (like radius for Circle, side for Square) and their own implementation of the calculateArea() method.

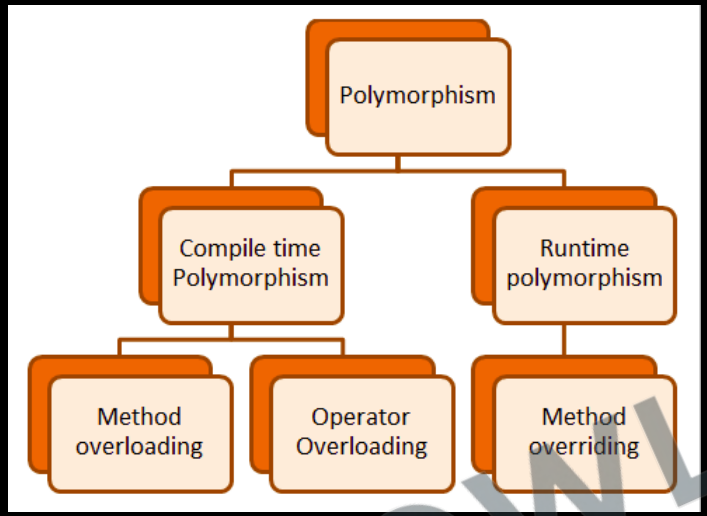
84. Create an interface Flyable with an abstract method fly(). Create an abstract class Bird that implements Flyable. Implement a subclass Eagle that extends Bird. Provide an implementation for the fly() method.

9.4 What is Polymorphism



1. Polymorphism is the ability of objects of different classes to respond to the same message (method call) in different ways.
2. Flexibility: Allows for writing more flexible and reusable code.
3. Simplicity: Enables developers to write more simple and readable code by using the same interface for different underlying data types.

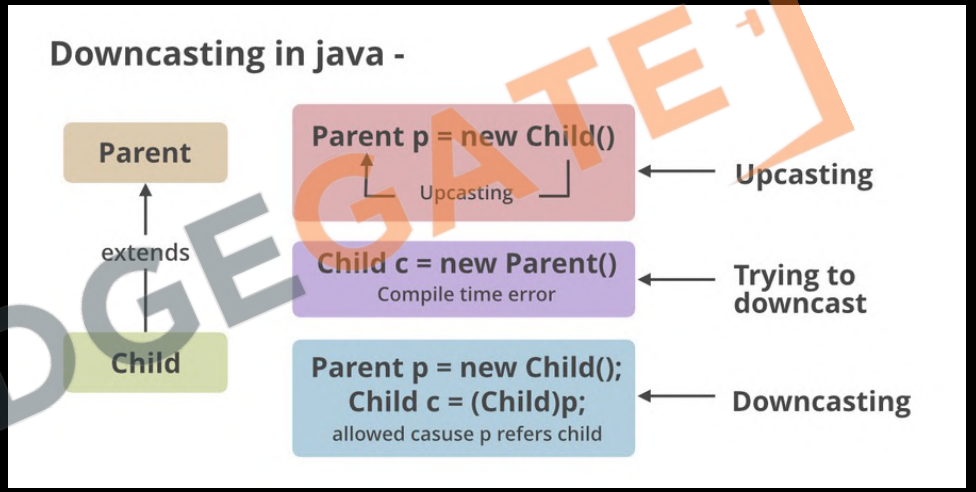
9.4 What is Polymorphism



1. Compile-Time Polymorphism: Achieved through method overloading or operator overloading.
2. Run-Time Polymorphism: Achieved through method overriding, where a subclass provides a specific implementation of a method already defined in its superclass.

9.5 References and Objects

1. Upcasting:
   - Converts subclass to superclass reference.
   - Automatic and safe.
   - Access only to superclass methods.
2. Downcasting:
   - Converts superclass to subclass reference.
   - Manual and risky, needs instanceof check.
   - Access to subclass-specific methods.
3. Usage:
   - Upcasting for generalization in methods.
   - Downcasting for specific subclass behaviors.



9.6 Method / Constructor Overloading

```java
// add() method is overloaded
public static int add(int a,int b) {
    return a + b; //adds integers
}

public static String add(String s1,String s2) {
    return s1.concat(s2); //concats strings
}
```

```java
class Dog{
    public void bark(){
        System.out.println("woof ");
    }
    //overloading method
    public void bark(int num){
        for(int i=0; i<num; i++)
            System.out.println("woof ");
    }
}
```

Same Method Name, Different Parameter

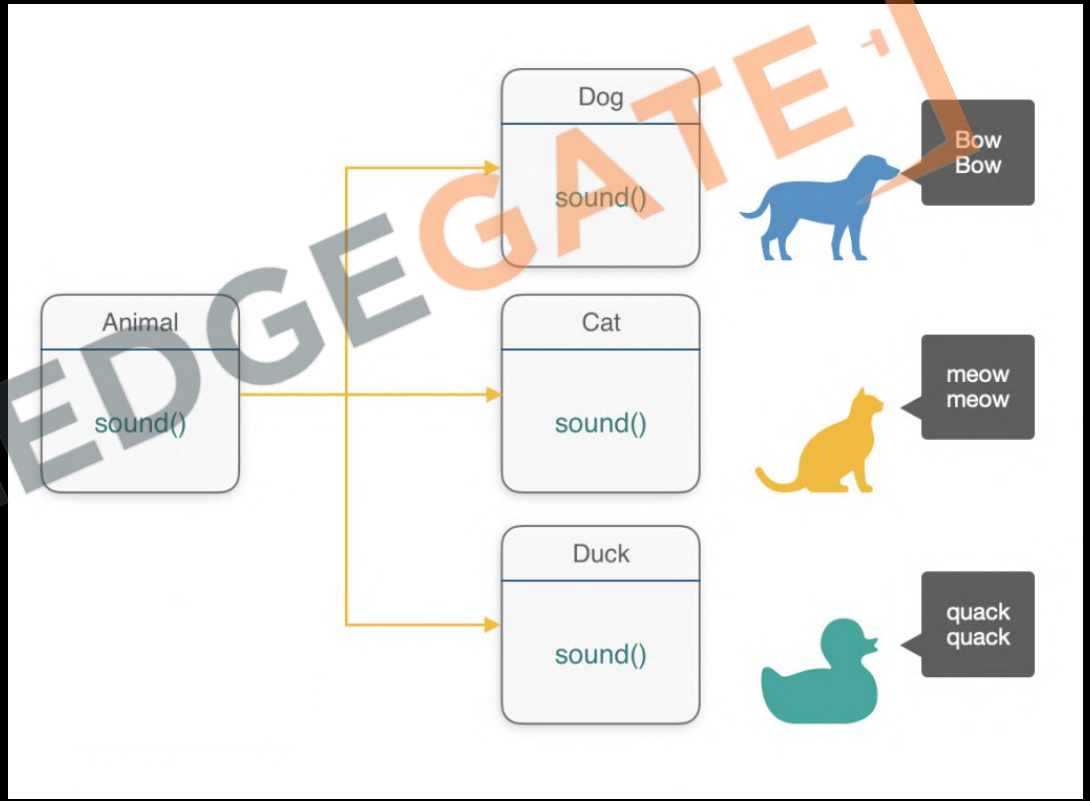
1. Method overloading occurs when multiple methods in the same class have the same name but different parameter lists.
2. Parameter Difference: Overloaded methods must differ in the number, type, or sequence of their parameters.
3. Return Type: Can vary between overloaded methods, but the return type alone does not distinguish them.
4. Compile-Time Polymorphism: It's a form of polymorphism that is resolved during compile time.

9.7 Super Keyword

1. Super can be used to refer immediate parent class instance varible.
2. Super can be used to invoke immediate parent class method.
3. Super () can be used to invoke immediate parent class constructor.

9.8 Method / Constructor Overriding

1. Method overriding occurs when a subclass provides a specific implementation for a method already defined in its superclass.
2. Run-Time Polymorphism: Overriding is a basis for runtime polymorphism, where the method call is determined by the object's type at runtime.
3. Superclass Reference: An overridden method can be called through a superclass reference holding a subclass object.



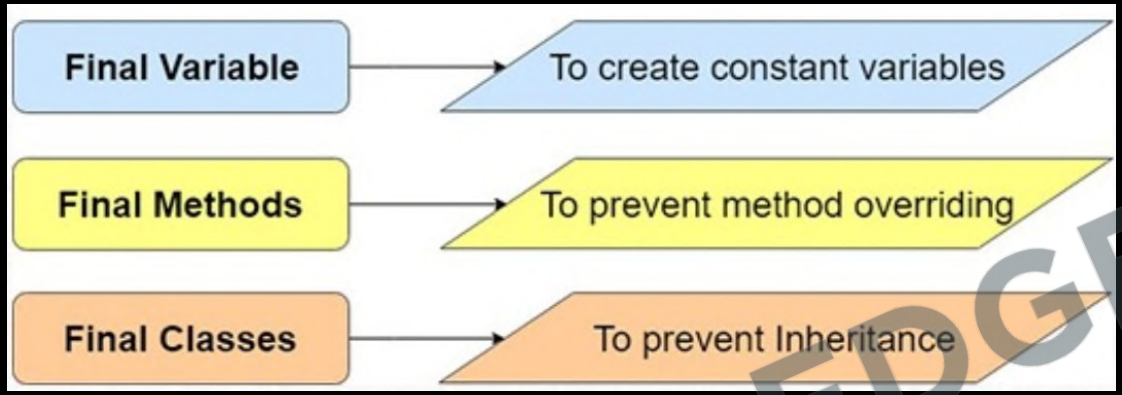
9.8 Method / Constructor Overriding

```
class Dog{
    public void bark(){
        System.out.println("woof ");
    }
}
class Hound extends Dog{
    public void sniff(){
        System.out.println("sniff ");
    }

    public void bark(){
        System.out.println("bowl");
    }
}
```

Same Method Name,
Same parameter

1. Same Signature: Overridden methods must have the same name, return type, and parameters as the method in the parent class.
2. Access Level: The access level cannot be more restrictive than the overridden method's access level.
3. **@Override Annotation**: This annotation is optional but helps to ensure that the method is correctly overridden.

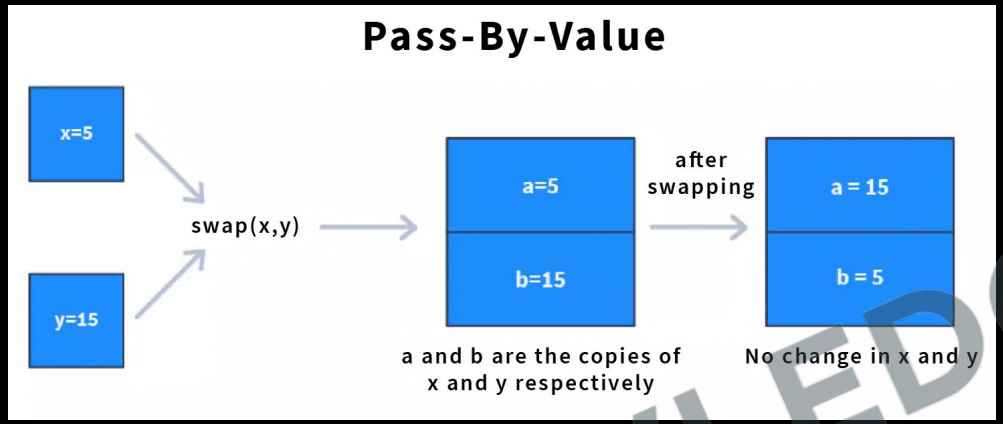
9.9 Final keyword revisited

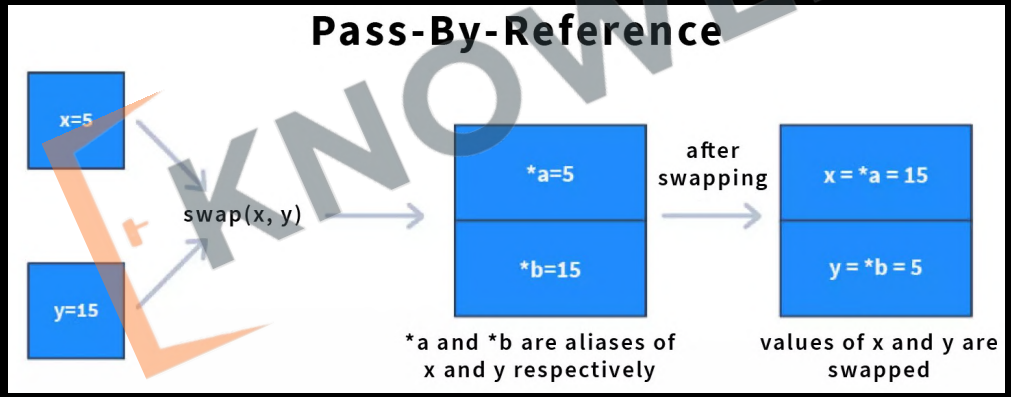


1. Variable becomes a constant, meaning its value cannot be changed once initialized.
2. Method: A final method cannot be overridden by subclasses.
3. Class: A final class cannot be subclassed, securing the class from being extended.
4. Efficiency: Using final can lead to performance optimization, as the compiler can make certain assumptions about final elements.
5. Null Safety: A final variable must be initialized before the constructor completes, reducing null pointer errors.
6. Immutable Objects: Helps in creating immutable objects in combination with private fields and no setter methods.

9.10 Pass by Value vs Pass by reference

9.10 Pass by Value vs Pass by reference





1. Variable Pass by Value:
   - Java's default method.
   - Copies argument's value to function's parameter.
   - Changes in function don't affect original variable.
2. Objects and References:
   - Java passes the reference's value for objects.
   - Modifications to objects in methods affect originals.
3. Primitive Types:
   - Always passed by value.
   - In-function changes don't impact originals.

CHALLENGE

85. In a class Calculator, create multiple add() methods that overload each other and can sum two integers, three integers, or two doubles. Demonstrate how each can be called with different numbers of parameters.

86. Define a base class Vehicle with a method service() and a subclass Car that overrides service(). In Car's service(), provide a specific implementation that calls super.service() as well, to show how overriding works.

Revision

1. What is Abstraction
2. Abstract Keyword
3. Interfaces
4. What is Polymorphism
5. References and Objects
6. Method / Constructor Overloading
7. Super Keyword
8. Method / Constructor Overriding
9. Final keyword revisited
10. Pass by Value vs Pass by reference.

Practice Exercise

Abstraction and Polymorphism

Answer in True/False

1. Abstraction hides the internal implementation details and shows only the functionality to the users.
2. An abstract class must contain at least one abstract method.
3. When you pass an object to a method, Java copies the reference to the object.
4. Overloaded methods must have different return types.
5. The 'super' keyword can be used to access methods of the superclass that are hidden by the subclass.
6. Method overriding is used to change the default behaviour of a method in the superclass.
7. The final keyword in Java is used to define constants.
8. Java supports pass by reference for primitive types.
9. An interface in Java can contain instance fields (variables).
10. A subclass can override a protected method from its superclass with a public access modifier.

Practice Exercise

Abstraction and Polymorphism

Answer in True/False

1. Abstraction hides the internal implementation details and shows only the functionality to the users. — True
2. An abstract class must contain at least one abstract method. — False
3. When you pass an object to a method, Java copies the reference to the object. — True
4. Overloaded methods must have different return types. — False
5. The 'super' keyword can be used to access methods of the superclass that are hidden by the subclass. — True
6. Method overriding is used to change the default behaviour of a method in the superclass. — True
7. The final keyword in Java is used to define constants. — True
8. Java supports pass by reference for primitive types. — False
9. An interface in Java can contain instance fields (variables). — False
10. A subclass can override a protected method from its superclass with a public access modifier. — True

KG Coding

Some Other One shot Video Links:

- Complete HTML
- Complete CSS
- Complete JavaScript
- Complete React and Redux
- One shot University Exam Series

http://www.kgcoding.in/

Our YouTube Channels

KG Coding Android App

KG Coding

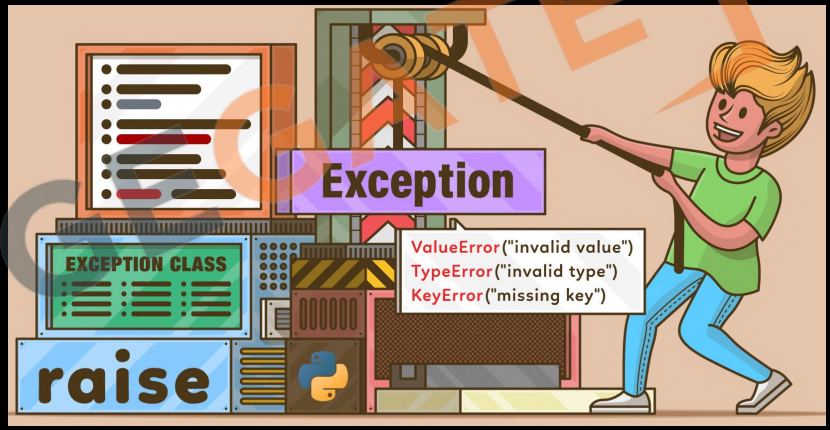
Knowledge GATE

KG Placement Prep

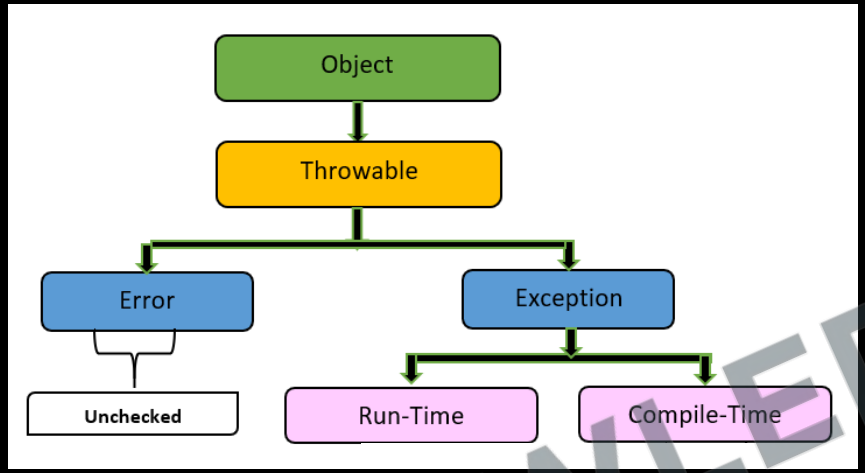
Sanchit Socket

10. Exception & File Handling

1. What is an Exception
2. Try-Catch
3. Types of Exception
4. Throw and Throws
5. Finally Block
6. Custom Exceptions
7. FileWriter class
8. FileReader class



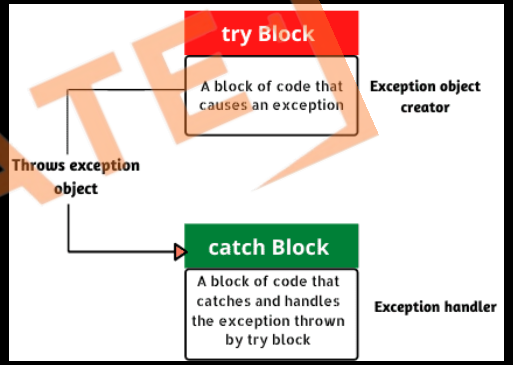
10.1 What is an Exception



1. In Java, an exception is a disruptive event that occurs during the execution of a program, interrupting its normal flow. It's an instance of a problem that arises while the program is running, such as arithmetic errors, null pointer accesses, or resource overflows.
2. Exceptions are objects in Java that encapsulate information about an error event, including its type and the state of the program when the error occurred.

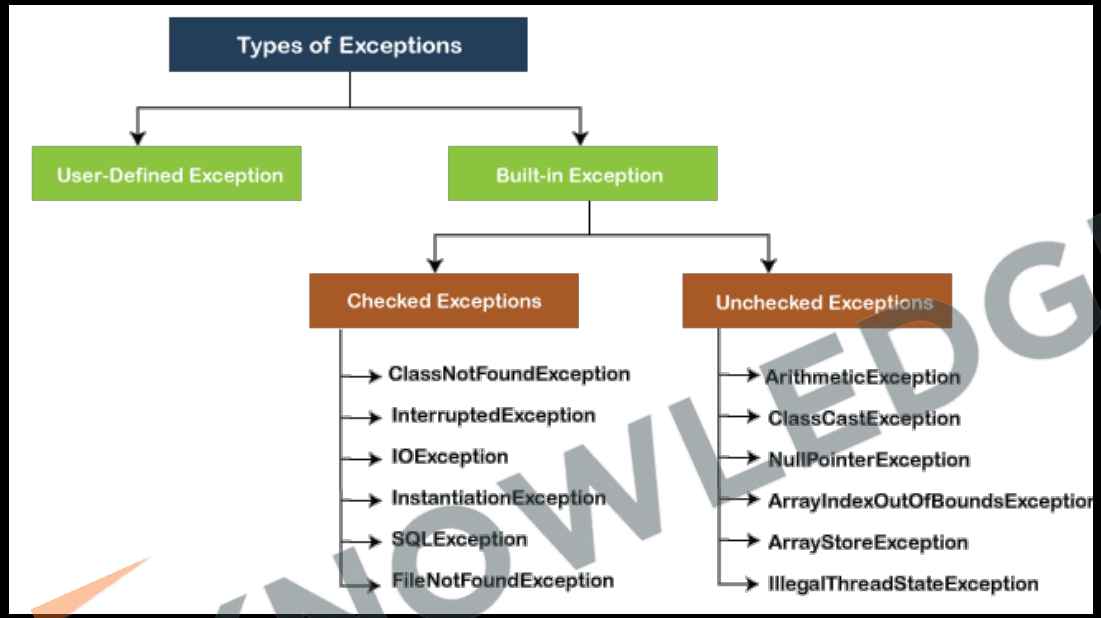
10.2 Try-Catch

```java
try{
    int num = 60/0;
    System.out.println(num);
} catch (ArrayIndexOutOfBoundsException e) {
    System.out.println("Array index exception: " + e.getStackTrace());
} catch (NumberFormatException | ArithmeticException e) {
    System.out.println("Multiple exceptions");
} catch (Exception e) {
    System.out.println("Last exception");
}
```



1. Try Block: Contains code that is susceptible to exceptions.
2. Catch Block: Follows the try block and handles the exceptions thrown by the try block.
3. When an exception occurs in the try block, the control is transferred to the catch block, where the exception is handled.

10.3 Types of Exceptions



1. Checked Exceptions: These are exceptions that must be either caught or declared in the method.
2. Unchecked Exceptions: These are exceptions that do not need to be explicitly handled.

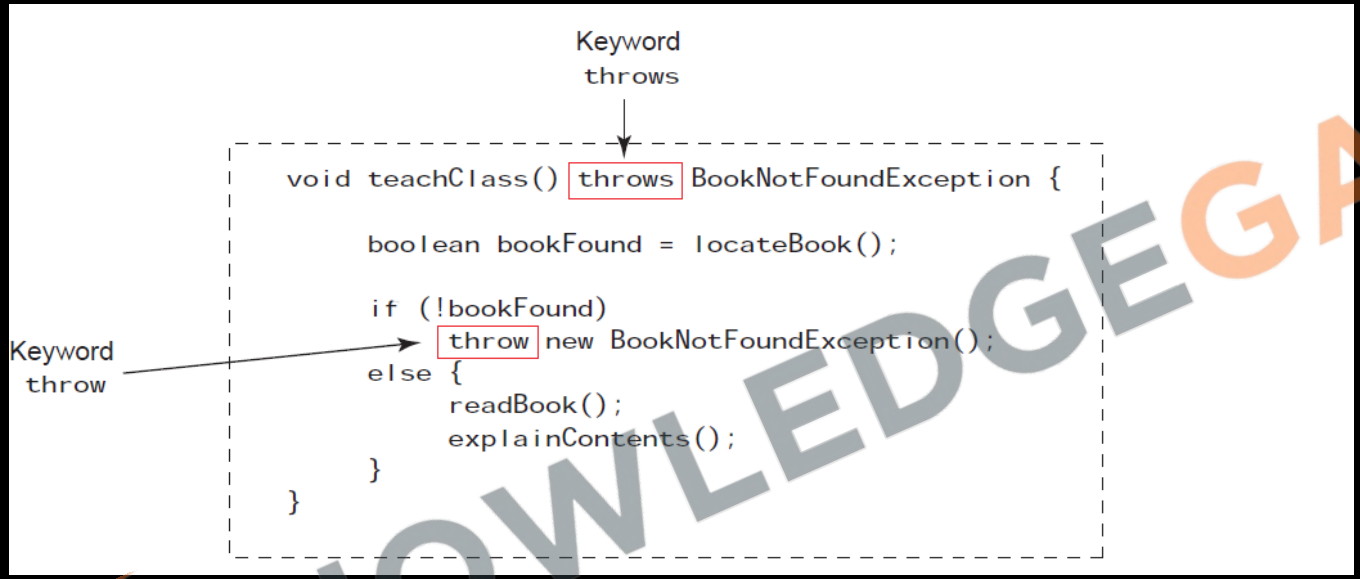
10.4 Throw and Throws

```
public void printName(String name)
        throws IllegalArgumentException {
    if (name.contains("-")) {
        throw new IllegalArgumentException("Name contains -");
    }
    System.out.println(name);
}
```


throws Keyword:

1. Declares that a method may throw one or more exceptions.
2. Used in the method signature to indicate that the method might throw exceptions of specified types.
3. A method declared with throws requires the calling method to handle or further declare the exception.

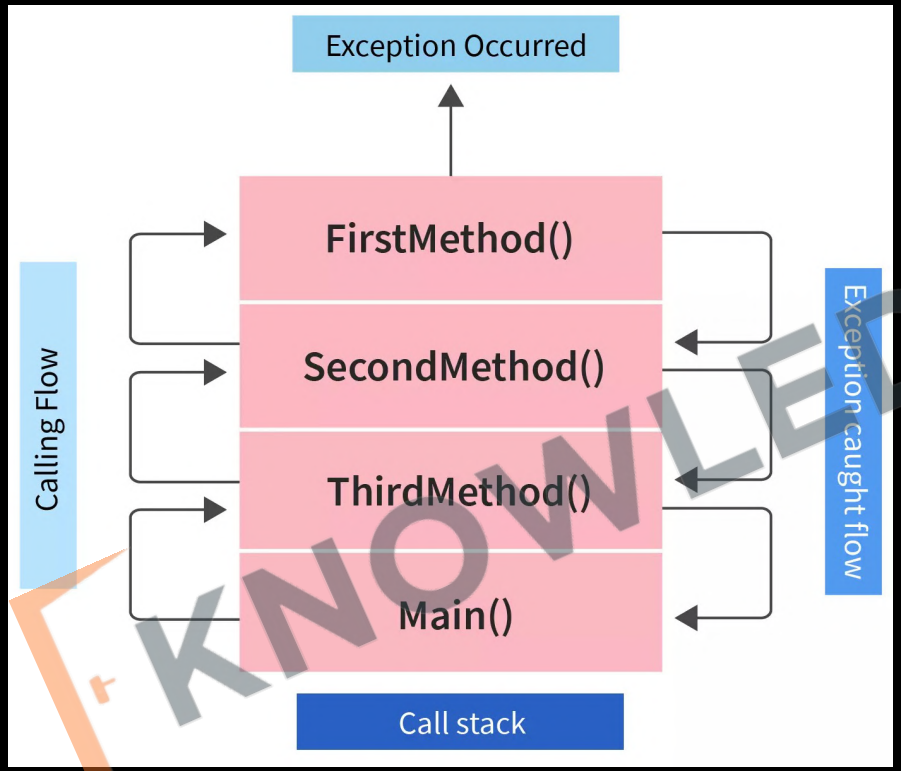
10.4 Throw and Throws



throw Keyword:

1. Used to explicitly throw an exception from any method or block of code.
2. You can throw either a new instance of an exception or an existing exception object using throw.
3. Example: throw new ArithmeticException("Division by zero");

10.4 Throw and Throws



10.4 Throw and Throws

throw	throws
Java **throw** keyword is used to explicitly throw an exception.	Java **throws** keyword is used to declare an exception.
Checked exception cannot be propagated using **throw**.	Checked exception can be propagated with **throws**.
If we see syntax wise, **throw** is followed by an instanceof Exception class Example : throw new NumberFormatException("The month entered, is invalid.");	If we see syntax wise, **throws** is followed by exception class names. Example : throws IOException,SQLException
The keyword **throw** is used inside method body.	**throws clause** is used in method declaration (signature).
By using **throw keyword** in java you cannot throw more than one exception. Example: throw new IOException("Connection failed!!")	By using **throws** you can declare multiple exceptions. Example: public void method() throws IOException,SQLException.

10.5 Finally Block

```
try {
    try {
        int result = 1 / 0;
    } catch (SomeException e) {
        System.out.println("Something caught");
    } finally {
        System.out.println("Not quite finally");
    }
} catch (ArithmeticException e) {
    System.out.println("ArithmeticException caught");
} finally {
    System.out.println("Finally");
}
```


1. Executes code after the try-catch blocks, used mainly for cleanup operations.
2. Always runs regardless of whether an exception is thrown or caught in the try-catch blocks.
3. Ideal for closing resources like files or database connections to prevent resource leaks.

10.6 Custom Exceptions

```
public class TemperatureException extends Exception {
    private double degrees;

    public TemperatureException(double degrees) {
        this.degrees = degrees;
    }

    public String getMessage() {
        return "The temperature (" + degrees
            + "C) isn't in the normal range.";
    }

    public double getDegrees() {
        return degrees;
    }
}
```

Override of Exception's getMessage() method

New method

1. Custom exceptions are user-defined exception classes that extend either Exception (for checked exceptions) or RuntimeException (for unchecked exceptions).
2. They are created to represent specific error conditions relevant to an application.

CHALLENGE

87. Arithmetic Exception Handling

Write a program that asks the user to enter two integers and then divides the first by the second. The program should handle any arithmetic exceptions that may occur (like division by zero) and display an appropriate message.

Key Points:

- Use Scanner to read user input.
- Implement a try-catch block to handle ArithmeticException.
- Display a user-friendly message if division by zero occurs.

10.7 FileWriter class

1. FileWriter is used for writing streams of characters to files.
2. It's a character-based stream, which means it's best used for writing text rather than binary data.
3. Constructors:
   - FileWriter(String fileName): Creates a FileWriter object given the name of the file to write to.
   - FileWriter(File file): Creates a FileWriter object given a File object.
4. Common Methods:
   - write(int c): Writes a single character.
   - write(char[] cbuf): Writes an array of characters.
   - write(String str): Writes a string.
   - flush(): Flushes the stream, ensuring all data is written out.
   - close(): Closes the stream, releasing any associated system resources.

10.7 FileWriter class

```java
public class FileWriterExample {
    new *
    public static void main(String[] args) {
        // Define the filename
        String fileName = "example.txt";

        // Create a FileWriter object
        try (FileWriter writer = new FileWriter(fileName)) {
            // Write a string to the file
            writer.write( str: "Hello, this is a test.");

            // Optionally, you can flush the writer
            writer.flush();

            System.out.println("Successfully written to the file.");
        } catch (IOException e) {
            System.out.println("An error occurred.");
            e.printStackTrace();
        }
    }
}
```

10.8 FileReader class

1. The FileReader class is used for reading streams of characters from files.
2. It's a character-based stream, meaning it reads characters (as opposed to bytes). This makes it suitable for reading text files.
3. Constructors:
   - FileReader(String fileName): Creates a FileReader object to read from a file with the specified name.
   - FileReader(File file): Creates a FileReader object to read from the specified File object.
4. Common Methods:
   - read(): Reads a single character and returns it as an integer. Returns -1 if the end of the stream is reached.
   - read(char[] cbuf): Reads characters into an array and returns the number of characters read.

```java
public class FileReaderExample {
    new *
    public static void main(String[] args) {
        // Define the filename
        String fileName = "example.txt";

        // Create a FileReader object
        try (FileReader reader = new FileReader(fileName)) {
            int character;

            // Read and display characters one by one
            while ((character = reader.read()) != -1) {
                System.out.print((char) character);
            }
        } catch (IOException e) {
            System.out.println("An error occurred.");
            e.printStackTrace();
        }
    }
}
```

CHALLENGE

88. File Not Found Exception Handling

Write a program to read a filename from the user and display its content. The program should handle the situation where the file does not exist.

Key Points:

- Use Scanner to read the filename from the user.
- Use FileReader to read the file content.
- Implement a try-catch block to handle FileNotFoundException.
- Display a message informing the user if the file is not found.

Revision

1. What is an Exception
2. Try-Catch
3. Types of Exception
4. Throw and Throws
5. Finally Block
6. Custom Exceptions
7. FileWriter class
8. FileReader class

Practice Exercise

Exception & File Handling

Answer in True/False

1. A method that throws a checked exception must declare the exception using the throws keyword in its signature.
2. It is possible to handle more than one type of exception in a single catch block.
3. The finally block is always executed, even if a return statement is encountered in the try or catch block.
4. If both try and catch blocks have return statements, the finally block will not execute.
5. You can throw an exception of any type, including custom exceptions, using the throw keyword.
6. If a try block does not throw any exception, the catch block is executed for cleanup purposes.
7. Unchecked exceptions are a direct subclass of Throwable.
8. A catch block can exist independently of a try block.
9. You can nest try blocks within another try block.
10. The throw keyword is used to propagate an exception up the call stack.

Practice Exercise

Exception & File Handling

Answer in True/False

1. A method that throws a checked exception must declare the exception using the throws keyword in its signature. — True
2. It is possible to handle more than one type of exception in a single catch block. — True
3. The finally block is always executed, even if a return statement is encountered in the try or catch block. — True
4. If both try and catch blocks have return statements, the finally block will not execute. — False
5. You can throw an exception of any type, including custom exceptions, using the throw keyword. — True
6. If a try block does not throw any exception, the catch block is executed for cleanup purposes. — False
7. Unchecked exceptions are a direct subclass of Throwable. — False
8. A catch block can exist independently of a try block. — False
9. You can nest try blocks within another try block. — True
10. The throw keyword is used to propagate an exception up the call stack. — True

KG Coding

Some Other One shot Video Links:

- Complete HTML
- Complete CSS
- Complete JavaScript
- Complete React and Redux
- One shot University Exam Series

http://www.kgcoding.in/

Our YouTube Channels

KG Coding Android App

KG Coding

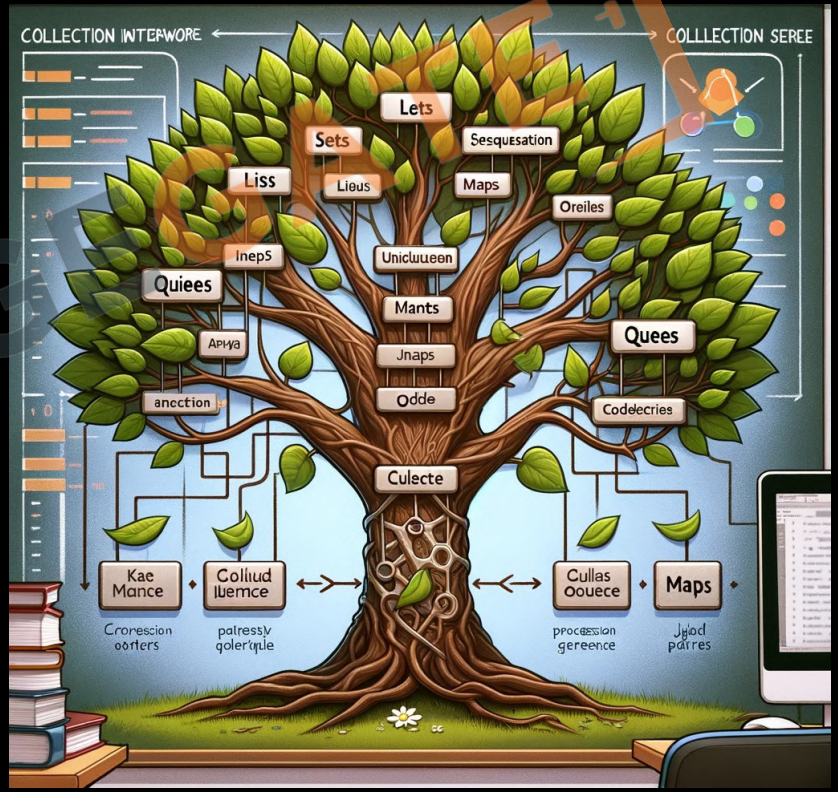
Knowledge GATE

KG Placement Prep

Sanchit Socket

11 Collections & Generics

1. Variable Arguments
2. Wrapper Classes & Autoboxing
3. Collections Library
4. List Interface
5. Queue Interface
6. Set Interface
7. Collections Class
8. Map Interface
9. Enums
10. Generics & Diamond Operators

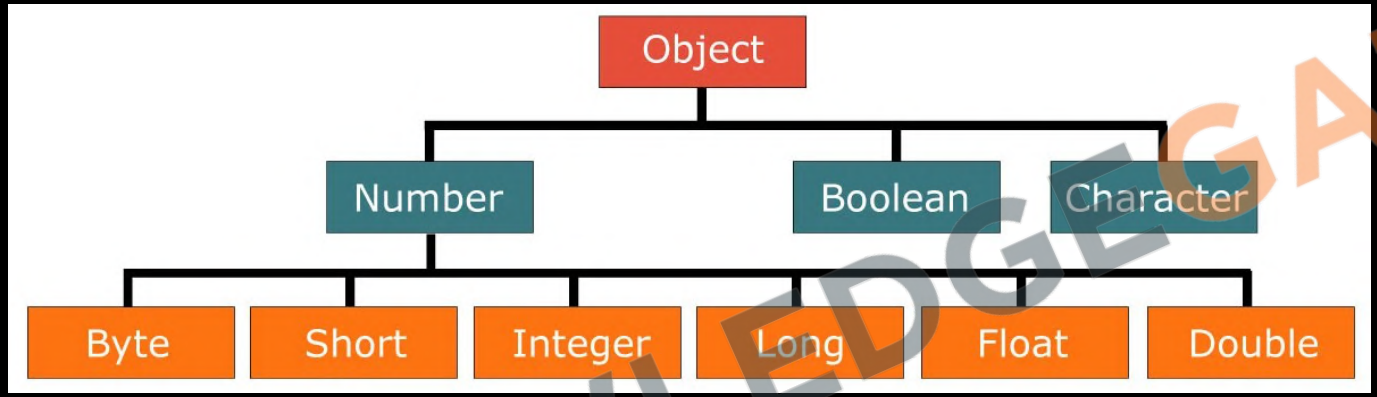


11.1 Variable Arguments

```java
public class MyClass {
    4 usages   new *
    static void printMany(String ...elements) {
        for (String element : elements) {
            System.out.printf(element);
        }
    }

    new *
    public static void main(String[] args) {
        printMany( ...elements: "one", "two", "three");
        printMany( ...elements: "one", "two");
        printMany();
        printMany(new String[]{"one", "two", "three"});
    }
}
```

1. Java's varargs allow methods to accept any number of arguments.
2. Declared using an ellipsis (...), e.g., void method(int... nums).
3. Internal Handling: Treated as arrays, e.g., int... nums is int[] nums.
4. Placement: Must be the last in the method's parameters.
5. Usage: Call with varying argument counts, e.g., method(1, 2) or method().
6. Introduced in: Java 5.

11.2 Wrapper Classes

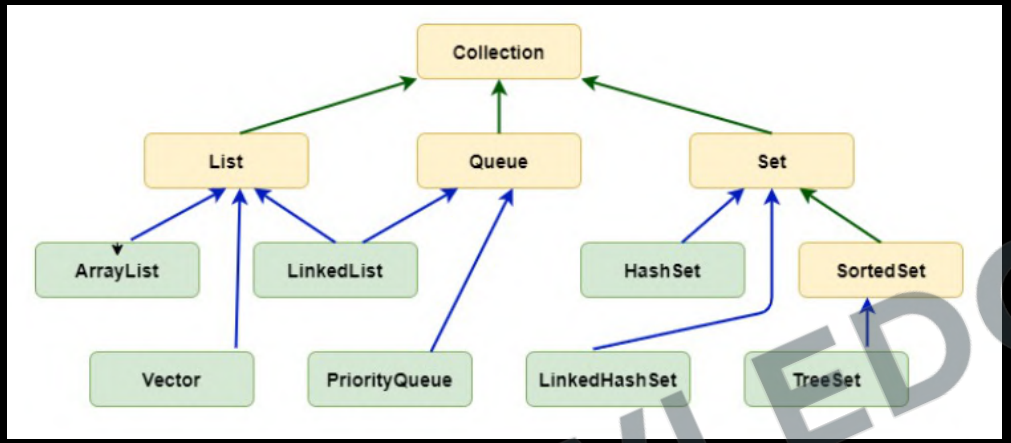


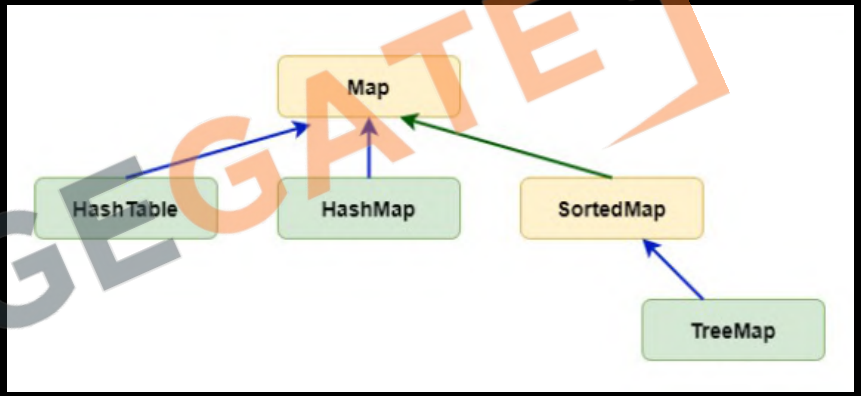
1. Provide a way to use primitive data types (int, char, boolean, etc.) as objects.
2. Automatic conversion between the primitive types and their corresponding wrapper classes.
3. Once created, the value of a wrapper object cannot be changed.
4. Utility Methods: Each wrapper class provides useful methods, like compareTo, valueOf, and parseXxx (e.g., parseInt for Integer).
5. Required for storing primitives in collection objects like ArrayList, HashMap, etc.
6. Allows assignment of null to primitive values when needed.

11.2 Autoboxing

1. Autoboxing: Automatic conversion of primitive types to their corresponding wrapper class objects.
2. Unboxing: Automatic conversion of wrapper class objects back to their respective primitive types.

11.3 Collections Library





1. Collection Interface: The root interface of the collection hierarchy. It declares basic operations like add, remove, clear, and size.
2. List Interface: An ordered collection. Lists can contain duplicate elements.
3. Set Interface: A collection that cannot contain duplicate elements.
4. Queue Interface: A collection used for holding elements in FIFO prior to processing.
5. Map Interface: Not a true Collection, but part of the Collections Framework. Maps store key-value pairs. Keys are unique, but different keys can map to the same value.

11.4 List Interface

1. An ordered collection (also known as a sequence).
2. Allows duplicate elements.
3. Elements can be accessed by their integer index.
4. Maintains the insertion order of elements.
5. Performance: Offers fast random access and quick iteration.
6. Capacity: Grows automatically as elements are added.
7. Preferred over arrays when the size is dynamic or unknown.

11.4 List Interface

1. add(E e): Appends the specified element
2. add(int index, E element): Inserts at specified position
3. remove(Object o): Removes the first occurrence of the specified element
4. remove(int index): Removes the element at the specified position
5. get(int index): Returns the element at the specified position
6. set(int index, E element): Replaces the element at the specified position
7. size(): Returns the number of elements
8. clear(): Removes all of the elements
9. contains(Object o): Returns `true` if the list contains the specified element.
10. indexOf(Object o): Returns the index of the first occurrence, or -1 if the list does not contain the element.

```java
//List Creation
List<Integer> list = new ArrayList<Integer>();
//List Addition
list.add(1);
list.add(2);
//Printing the List
System.out.println(list);
```

11.5 Queue Interface

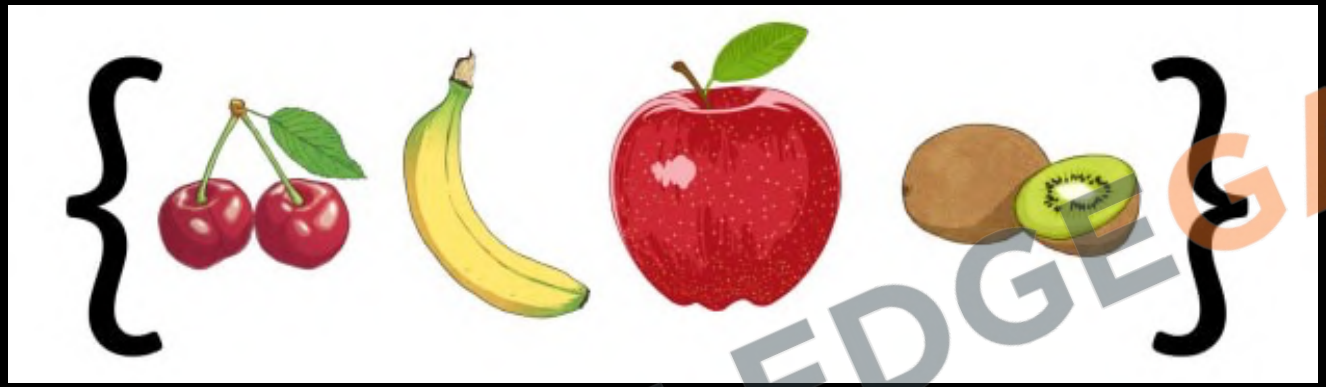
1. It's a collection designed for holding elements prior to processing.
2. Ordering: Typically, it orders elements in a FIFO (First-In-First-Out) manner.
3. End Points: Offers two ends - one for insertion (tail) and the other for removal (head).

11.5 Queue Interface

1. add(E e): Inserts the specified element into the queue. Throws an exception if the element cannot be added.
2. offer(E e): Inserts the specified element into the queue. Returns false if the element cannot be added.
3. remove(): Retrieves and removes the head of the queue. Throws an exception if the queue is empty.
4. poll(): Retrieves and removes the head of the queue, or returns null if the queue is empty.
5. element(): Retrieves, but does not remove, the head of the queue. Throws an exception if the queue is empty.
6. peek(): Retrieves, but does not remove, the head of the queue, or returns null if the queue is empty.

```java
// Creating a Queue using LinkedList
Queue<String> queue = new LinkedList<>();

// Adding elements (offer method can also be used)
queue.add("First");
queue.add("Second");

// Displaying the head of the queue
System.out.println("Head of queue: " + queue.peek());

// Removing elements from the queue
while (!queue.isEmpty()) {
    System.out.println("Removed: " + queue.poll());
}
```

11.6 Set Interface



1. Unique Elements: Does not allow duplicate elements.
2. Unordered Collection: it does not guarantee any specific ordering of elements.
3. No Positional Access: Unlike lists, it doesn't support indexing-based access to elements.
4. Implementation: Common implementations include HashSet, LinkedHashSet, and TreeSet.

11.6 Set Interface

```java
// Creating a Set
Set<String> set = new HashSet<>();

// Adding elements
set.add("Apple");

// Attempting to add a duplicate element
boolean isAdded = set.add("Apple"); // This will be false
System.out.println("Apple added again: " + isAdded);

// Checking if a specific element is in the set
boolean containsOrange = set.contains("Orange");
System.out.println("Contains Orange: " + containsOrange);
```


1. add(E e): Adds the specified element to the set
2. remove(Object o): Removes the specified element from the set
3. contains(Object o): Checks if the set contains the specified element.
4. size(): Returns the number of elements in the set.
5. isEmpty(): Checks if the set is empty.

11.7 Collections Class

```java
// Creating a list
List<Integer> list = new ArrayList<>();
Collections.addAll(list, ...elements: 5, 1, 8, 3, 2);

// Sorting the list
Collections.sort(list);
System.out.println("Sorted list: " + list);

// Finding max and min in the list
int max = Collections.max(list);
int min = Collections.min(list);
System.out.println("Max: " + max + ", Min: " + min);
```

```java
// Reversing the list
Collections.reverse(list);
System.out.println("Reversed list: " + list);

// Searching in the list (List must be sorted)
Collections.sort(list);
int index = Collections.binarySearch(list, key: 3);
System.out.println("Index of 3: " + index);

// Creating an immutable list
List<Integer> unmodifiableList = Collections.unmodifiableList(list);
System.out.println("Unmodifiable list: " + unmodifiableList);
```


1. Offers methods like sort to sort lists.
2. Provides methods like binarySearch for searching sorted lists.
3. Allows reversing the order of elements in a list with reverse.
4. Can shuffle the elements of a list randomly using shuffle.
5. Creates unmodifiable collections using methods like unmodifiableList, etc.
6. Methods like singletonList, create immutable collections with a single element.
7. The copy method is used to copy all elements from one list to another.

CHALLENGE

89. Write a method concatenate Strings that takes variable arguments of String type and concatenates them into a single string.
90. Write a program that sorts a list of String objects in descending order using a custom Comparator.
91. Use the Collections class to count the frequency of a particular element in an ArrayList.
92. Write a method that swaps two elements in an ArrayList, given their indices.
93. Create a program that reverses the elements of a List and prints the reversed list.
94. Create a PriorityQueue of a custom class Student with attributes name and grade. Use a comparator to order by grade.
95. Write a program that takes a string and returns the number of unique characters using a Set.

11.8 Map Interface

1. Stores data as key-value pairs.
2. Each key can map to at most one value.
3. Keys are unique, but multiple keys can map to the same value.
4. It is part of the Collections Framework but does not extend the Collection interface.

11.8 Map Interface

```java
// Creating a Map
Map<String, Integer> map = new HashMap<>();

// Adding key-value pairs to the Map
map.put("Apple", 10);

// Accessing a value
Integer appleCount = map.get("Apple");
System.out.println("Apples count: " + appleCount);

// Checking if a key exists
if (map.containsKey("Banana")) {
    System.out.println("Banana is in the map");
}

// Removing a key-value pair
map.remove( key: "Orange");
```

1. put(K key, V value): Associates the specified value with the specified key in the map.
2. get(Object key): Returns the value to which the specified key is mapped, or null if the map contains no mapping for the key.
3. remove(Object key): Removes the mapping for a key from the map if it is present.
4. containsKey(Object key): Checks if the map contains a mapping for the specified key.
5. keySet(): Returns a Set view of the keys contained in the map.
6. values(): Returns a Collection view of the values contained in the map.

11.9 Enums

```java
enum TrafficSignal{
    RED("stop"), GREEN("start"), ORANGE("slow down");

    private String action;

    public String getAction(){
        return this.action;
    }
    private TrafficSignal(String action){
        this.action = action;
    }
}

// String to Enum using valueOf
TrafficSignal signal = TrafficSignal.valueOf("RED");
signal = TrafficSignal.valueOf("GREEN"); //OK
signal = TrafficSignal.valueOf("Green"); //Not Ok
```

1. Enums in Java: Special types for fixed sets of constants like days, colors.
2. Declaration: Use enum keyword, e.g., enum Color { RED, GREEN, BLUE; }.
3. Access: Access constants with dot syntax, e.g., Color.RED.
4. Features: Type-safe, readable, can have methods and fields.
5. Usage: Useful in switch statements and iterating with values() method.

11.10 Generics & Diamond Operators

```java
class SpecificClass {
    2 usages
    private String thing;
    no usages   new *
    public String getThing() {
        return thing;
    }
    no usages   new *
    public void setThing (String thing) {
        this.thing = thing;
    }
}
```

```java
class GenericClass<T> {
    2 usages
    private T thing;
    no usages   new *
    public T getThing() {
        return thing;
    }
    no usages   new *
    public void setThing(T thing) {
        this.thing = thing;
    }
}
```

1. They allow you to write flexible and reusable code by enabling types (classes and interfaces) to be parameters when defining classes, interfaces, and methods.
2. Generics provide compile-time type safety by allowing you to enforce that certain objects are of a specific type.
3. With generics, you don't need to cast objects because the type is known.
4. Generics are denoted by angle brackets <>, e.g., List<String> means a list of strings.
5. Diamond Operator: Introduced in Java 7, the diamond operator <> allows you to infer the type parameter from the context, simplifying instantiation of generic classes.

CHALLENGE

96. Create an enum called Day that represents the days of the week. Write a program that prints out all the days of the week from this enum.
97. Enhance the Day enum by adding an attribute that indicates whether it is a weekday or weekend. Add a method in the enum that returns whether it's a weekday or weekend, and write a program to print out each day along with its type.
98. Create a Map where the keys are country names (as String) and the values are their capitals (also String). Populate the map with at least five countries and their capitals. Write a program that prompts the user to enter a country name and then displays the corresponding capital, if it exists in the map.

Revision

1. Variable Arguments
2. Wrapper Classes & Autoboxing
3. Collections Library
4. List Interface
5. Queue Interface
6. Set Interface
7. Collections Class
8. Map Interface
9. Enums
10. Generics & Diamond Operators

Practice Exercise

Collections & Generics

Answer in True/False

1. Variable arguments can be used to pass an arbitrary number of values to a method.
2. Autoboxing is the process where primitive types are automatically converted into their corresponding wrapper class objects by the Java compiler.
3. Every class in the Collections Library implements the Collection interface.
4. Lists guarantee the order of insertion for the elements they contain.
5. Sets in Java inherently maintain the elements in sorted order.
6. In a Map, you can have multiple entries with the same value but not with the same key.
7. Enums in Java can have constructors, methods, variables, and can implement interfaces.
8. With the diamond operator, you do not need to specify the type on the right-hand side of a statement when initializing an object.
9. The List interface provides methods for inserting elements at a specific position in the list.
10. Wrapper classes in Java, such as Integer and Double, can be extended like any other class.

Practice Exercise

Collections & Generics

Answer in True/False

1. Variable arguments can be used to pass an arbitrary number of values to a method. — True
2. Autoboxing is the process where primitive types are automatically converted into their corresponding wrapper class objects by the Java compiler. — True
3. Every class in the Collections Library implements the Collection interface. — False
4. Lists guarantee the order of insertion for the elements they contain. — True
5. Sets in Java inherently maintain the elements in sorted order. — False
6. In a Map, you can have multiple entries with the same value but not with the same key. — True
7. Enums in Java can have constructors, methods, variables, and can implement interfaces. — True
8. With the diamond operator, you do not need to specify the type on the right-hand side of a statement when initializing an object. — True
9. The List interface provides methods for inserting elements at a specific position in the list. — True
10. Wrapper classes in Java, such as Integer and Double, can be extended like any other class. — False

KG Coding

Some Other One shot Video Links:

- Complete HTML
- Complete CSS
- Complete JavaScript
- Complete React and Redux
- One shot University Exam Series

http://www.kgcoding.in/

Our YouTube Channels

KG Coding Android App

KG Coding

Knowledge GATE

KG Placement Prep

Sanchit Socket

12 Multi threading & Executor Service

1. Intro to Multi-threading
2. Creating a Thread
3. States of a Thread
4. Thread Priority
5. Join Method
6. Synchronize keyword
7. Thread Communication
8. Intro to Executor Service
9. Multiple Threads with Executor
10. Returning Futures

12.1 What is a Thread

1. What is a Thread: A thread in Java is a small part of a program that can run at the same time as other parts.
2. Purpose: Threads help a program do many things at once, like handling many users or doing different tasks simultaneously.
3. Creating Threads: You can make a thread by using the Thread class or the Runnable interface.
4. Using Threads: Use threads for tasks that can happen at the same time, like managing many requests or splitting up a big job.
5. Thread Talk: Threads can talk to each other using wait(), notify(), and notifyAll() to coordinate their work.

12.1 Need of Multi-threading

```java
// First Task
for (int i = 1; i <= 1000; i++) {
    System.out.printf("%d:* ", i);
}
System.out.println("\nFirst Task Done");

// Second Task
for (int i = 1; i <= 1000; i++) {
    System.out.printf("%d:& ", i);
}
System.out.println("\nSecond Task Done");

// Third Task
for (int i = 1; i <= 1000; i++) {
    System.out.printf("%d:$ ", i);
}
System.out.println("\nThird Task Done");
```

1. Tasks might be very important
2. Tasks are independent of each other
3. A Multi-core CPU is sitting idle most of the time
4. A big task can be divided into smaller parts
5. Making your code responsive

12.2 Creating a Thread

(Extending Thread Class)

```
// Step 1: Define a Class that Extends Thread
4 usages
public class PrintTask extends Thread {
    // Step 2: Override the run() Method
    no usages
    public void run() {
        // First Task
        for (int i = 1; i <= 1000; i++) {
            System.out.printf("%d:%c ", i, targetChar);
        }
        System.out.printf("\n%c Task Done\n", targetChar);
    }
    3 usages
    private final char targetChar;
    2 usages
    public PrintTask(char targetChar) {
        this.targetChar = targetChar;
    }
}
```

```
public static void main(String... args) {
    // Step 3: Create an Instance of Your Class
    PrintTask t1 = new PrintTask( targetChar: '*');
    t1.start(); // Start the first thread

    PrintTask t2 = new PrintTask( targetChar: '$');
    t2.start(); // Start the second thread
}
```

In the main method, two threads (t1 and t2) are created and started. They will execute independently and print their values.

12.2 Creating a Thread

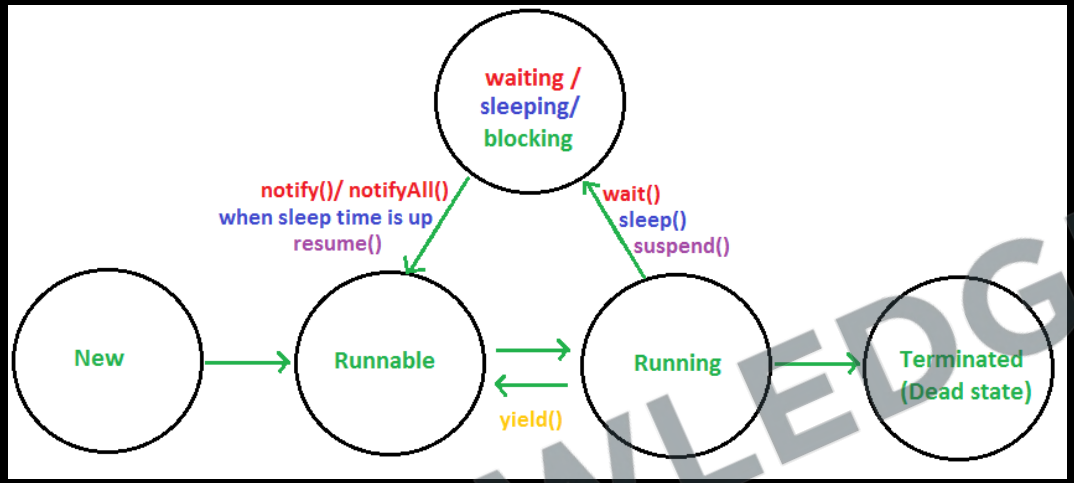
(Creating Runnables)

```java
// Step 1: Define a Class that implements Runnable
4 usages
public class PrintRunnable implements Runnable {
    // Step 2: Override the run() Method
    no usages
    @Override
    public void run() {
        // First Task
        for (int i = 1; i <= 1000; i++) {
            System.out.printf("%d:%c ", i, targetChar);
        }
        System.out.printf("\n%c Task Done\n", targetChar);
    }
    3 usages
    private final char targetChar;
    2 usages
    public PrintRunnable(char targetChar) {
        this.targetChar = targetChar;
    }
}
```

```java
public static void main(String... args) {
    // Step 3: Create an Instance of Your Class
    PrintRunnable t1 = new PrintRunnable( targetChar: '*');
    // Step4: Wrap your class with a thread
    new Thread(t1).start(); // Start the first thread

    PrintRunnable t2 = new PrintRunnable( targetChar: '$');
    new Thread(t1).start(); // Start the second thread
}
```

In the main method, two threads (t1 and t2) are created and started. They will execute independently and print their values.

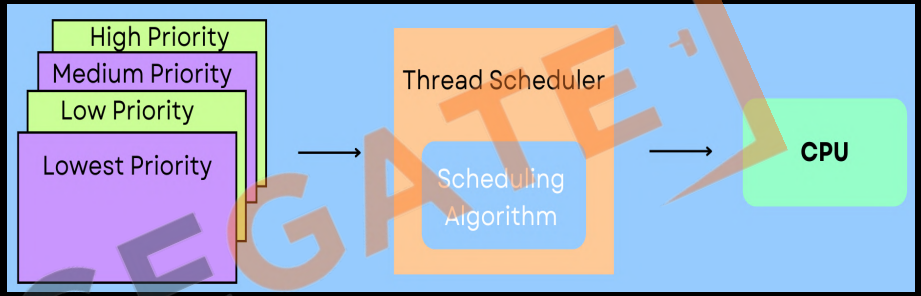
12.3 States of a Thread



1. New: Thread is created but not started.
2. Runnable: Thread is ready or running.
3. Running: Thread is actively executing tasks.
4. Blocked/Waiting: Thread is alive but not active because it's waiting for resources or other threads.
5. Terminated: Thread has finished or stopped running.

12.4 Thread Priority

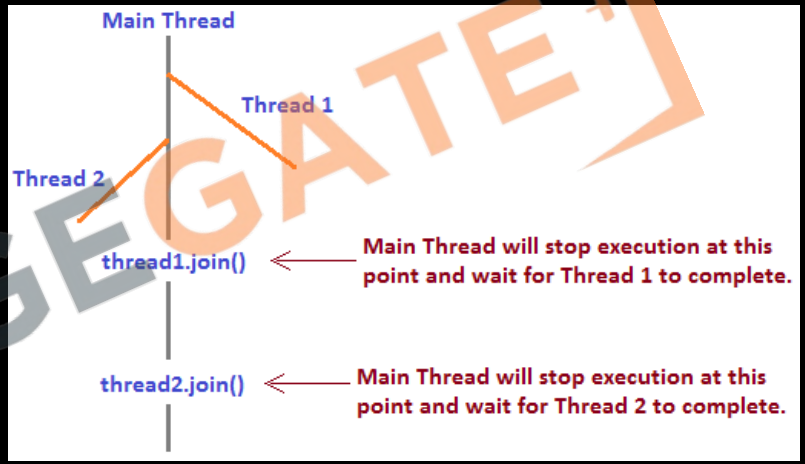
```java
class MyThread extends Thread {
    no usages
    public void run() {
        Thread current = Thread.currentThread();
        System.out.printf("Running thread name: %s\n",
                current.getName());
        System.out.printf("Running thread priority: %s\n",
                current.getPriority());
    }
}

public class ThreadPriority {
    public static void main(String args[]) {
        MyThread t1 = new MyThread();
        MyThread t2 = new MyThread();
        t1.setPriority(Thread.MIN_PRIORITY); // Setting priority to 1
        t2.setPriority(Thread.MAX_PRIORITY); // Setting priority to 10

        t1.setName("Thread-1");
        t2.setName("Thread-2");
        t1.start();
        t2.start();
    }
}
```



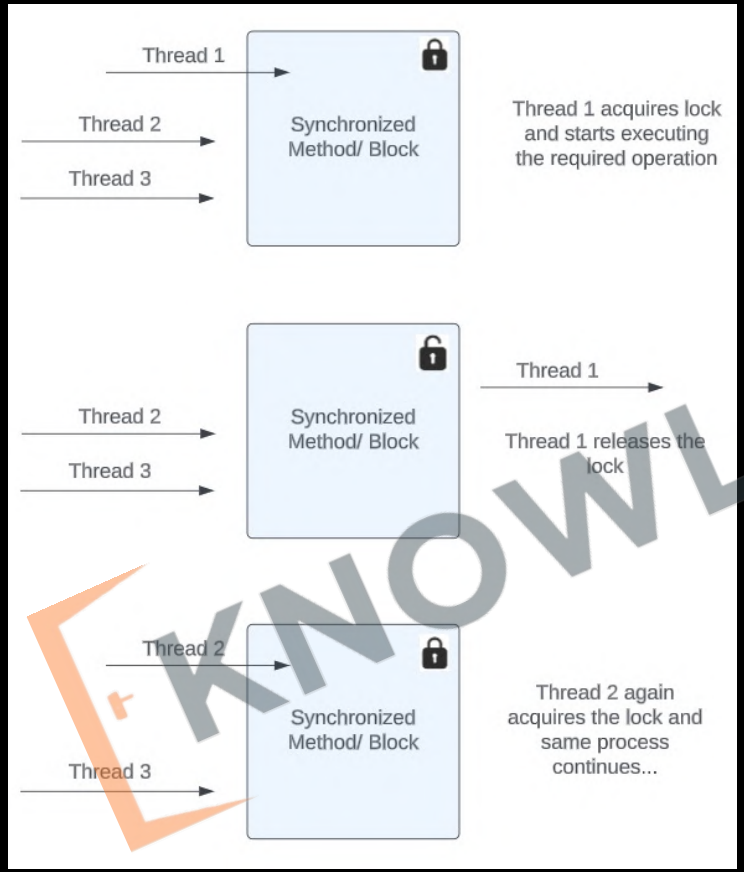
1. Priority Levels: Java threads have priority levels from 1 (lowest) to 10 (highest), with a default value of 5.
2. Influence on Execution: A thread's priority suggests the importance of a thread to the scheduler, though it doesn't guarantee the order of execution.
3. Set and Get Priority: Use setPriority(int) to change a thread's priority and getPriority() to retrieve it.

12.5 Join Method

1. Purpose of join: The join method is used to make the calling thread wait until the thread on which join has been called completes its execution.
2. Synchronization of Threads: join helps in synchronizing multiple threads, ensuring that a thread completes its execution before the next steps in the calling thread proceed.
3. Overloaded Versions: join comes in three versions:
   - join(): Waits indefinitely until the thread on which it's called finishes.
   - join(long millis): Waits for the thread to die for the specified milliseconds.
   - join(long millis, int nanos): Waits for the thread to die for the specified milliseconds plus nanoseconds.



12.6 Synchronize keyword



1. Mutual Exclusion: The synchronize keyword in Java ensures that only one thread can execute a block of code at a time, providing mutual exclusion and preventing race conditions.
2. Object Lock: When a thread enters a synchronized block or method, it acquires a lock on the object or class, depending on whether the method is an instance method or a static method.
3. Visibility: It ensures that changes made by one thread to shared data are visible to other threads.

12.6 Synchronize keyword

```
class Counter {
    2 usages
    private int count = 0;
    // Synchronized method to increment the counter
    1 usage
    public synchronized void increment() {
        count++;
    }
    // Method to get the current count
    1 usage
    public int getCount() {
        return count;
    }
}
4 usages
class SynchronizedThread extends Thread {
    2 usages
    private Counter counter;
    2 usages
    public SynchronizedThread(Counter counter) {
        this.counter = counter;
    }
    no usages
    public void run() {
        for (int i = 0; i < 1000; i++) {
            counter.increment();
        }
    }
}
```

```
public class SynchronizedExample {
    public static void main(String[] args) throws InterruptedException {
        Counter counter = new Counter();
        SynchronizedThread t1 = new SynchronizedThread(counter);
        SynchronizedThread t2 = new SynchronizedThread(counter);

        t1.start();
        t2.start();

        t1.join();
        t2.join();
        System.out.println("Final count is: " + counter.getCount());
    }
}
```

12.7 Thread Communication

1. sleep(long millis): Causes the currently executing thread to sleep (temporarily cease execution) for the specified number of milliseconds.
2. yield(): Causes the currently executing thread to pause and allow other threads to execute. It's a way of suggesting that other threads of the same priority can run.
3. wait(): Causes the current thread to wait until another thread invokes the notify() or notifyAll() method for this object. It releases the lock held by this thread.
4. notify(): Wakes up a single thread that is waiting on the object's monitor. If any threads are waiting, one is chosen to be awakened.
5. notifyAll(): Wakes up all threads that are waiting on the object's monitor.

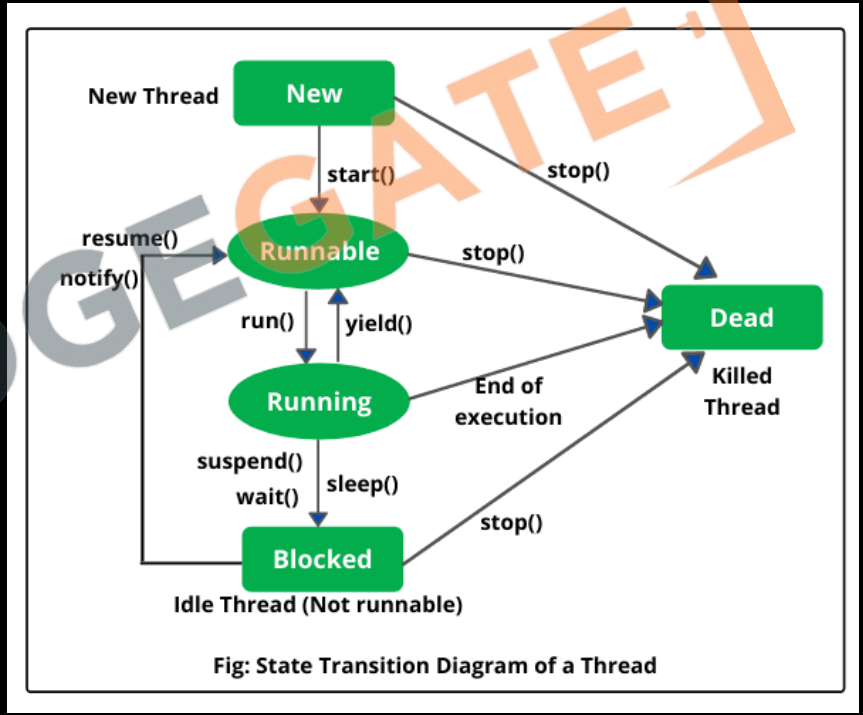
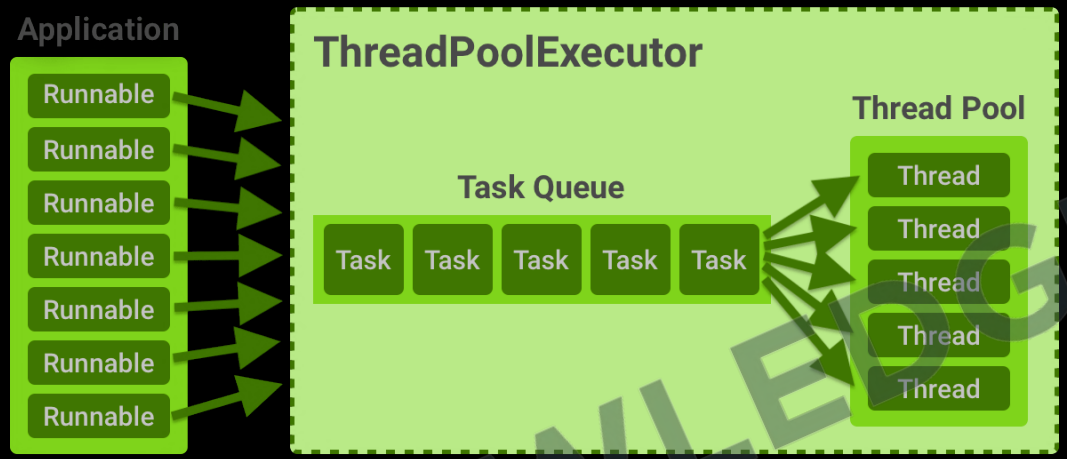


Fig: State Transition Diagram of a Thread

CHALLENGE

99. Write a program that creates two threads. Each thread should print "Hello from Thread X", where X is the number of the thread (1 or 2), ten times, then terminate.
100. Write a program that starts a thread and prints its state after each significant event (creation, starting, and termination). Use Thread.sleep() to simulate long-running tasks and Thread.getState() to print the thread's state.
101. Create three threads. Ensure that the second thread starts only after the first thread ends and the third thread starts only after the second thread ends using the join method. Each thread should print its start and end along with its name.
102. Simulate a traffic signal using threads. Create three threads representing three signals: RED, YELLOW, and GREEN. Each signal should be on for a certain time, then switch to the next signal in order. Use sleep for timing and synchronize to make sure only one signal is active at a time.

12.8 Intro to Executor Service



1. Purpose: ExecutorService is a framework provided by the Java Concurrency API to manage and execute submitted tasks without the need to manually manage thread life cycles.
2. Thread Pool Management: ExecutorService efficiently reuses a fixed pool of threads to execute tasks, thereby improving performance by reducing the overhead of thread creation, especially for short-lived asynchronous tasks.

12.8 Intro to Executor Service

```java
// Step 1: Define a Class that implements Runnable
5 usages
public class PrintRunnable implements Runnable {
    // Step 2: Override the run() Method
    no usages
    @Override
    public void run() {
        // First Task
        for (int i = 1; i <= 1000; i++) {
            System.out.printf("%d:%c ", i, targetChar);
        }
        System.out.printf("\n%c Task Done\n", targetChar);
    }
    3 usages
    private final char targetChar;
    3 usages
    public PrintRunnable(char targetChar) {
        this.targetChar = targetChar;
    }
}
```

```java
public class SingleThreadExecutorExample {
    public static void main(String[] args) {
        // Create a single-threaded executor
        ExecutorService executor = Executors.newSingleThreadExecutor();
        // Define a task (a Runnable)
        Runnable task = new PrintRunnable( targetChar: 'E');

        // Submit the task to the executor
        executor.submit(task);
        executor.shutdown();
    }
}
```

12.9 Multiple Threads with Executor

```java
public class PrintRunnable implements Runnable {
    // Step 2: Override the run() Method
    no usages
    @Override
    public void run() {
        // Task
        String threadName = Thread.currentThread().getName();
        System.out.printf("Executing task with char %c on" +
                "| Thread: %s \n", targetChar, threadName);
        for (int i = 1; i <= 1000; i++) {
            System.out.printf("%d:%c ", i, targetChar);
        }
        System.out.printf("\n%c Task Done\n", targetChar);
    }
    4 usages
    private final char targetChar;
    3 usages
    public PrintRunnable(char targetChar) {
        this.targetChar = targetChar;
    }
}
```

```java
public class MultiThreadExecutorExample {
    public static void main(String[] args) throws InterruptedException {
        // Create a single-threaded executor
        ExecutorService executor = Executors.newFixedThreadPool( nThreads: 3);

        // Define a task (a Runnable)
        for (int i = 0; i < 5; i++) {
            int taskNumber = i + 1;
            Runnable task = new PrintRunnable((char)taskNumber);
            // Submit the task to the executor
            executor.submit(task);
        }

        executor.shutdown();

        // Wait for all tasks to finish executing or timeout after 10 seconds
        if (!executor.awaitTermination( l: 10, TimeUnit.SECONDS)) {
            // Try to stop all actively executing tasks
            executor.shutdownNow();
        }
    }
}
```

12.10 Returning Futures

```java
// Create an executor with a fixed thread pool of 2 threads
ExecutorService executor = Executors.newFixedThreadPool( nThreads: 2);
// Submit the task to the executor and get a Future object
Future<String> future = executor.submit(task);

try {
    // Get the result from the Future object.
    String result = future.get();
    System.out.println("Result from future: " + result);
} catch (ExecutionException | InterruptedException e) {
    // Handle the interruption during the get
    Thread.currentThread().interrupt();
    System.out.println("Task was interrupted");
}

// Shut down the executor
executor.shutdown();
```

```java
// Define a callable task that returns a result
Callable<String> task = new Callable<String>() {
    @Override
    public String call() throws Exception {
        // Simulate task execution time
        TimeUnit.SECONDS.sleep( timeout: 1);
        return "Result from task";
    }
};
```

CHALLENGE

103. Write a program that creates a single-threaded executor service. Define and submit a simple Runnable task that prints numbers from 1 to 10. After submission, shut down the executor.
104. Create a fixed thread pool with a specified number of threads using Executors.newFixedThreadPool(int). Submit multiple tasks to this executor, where each task should print the current thread's name and sleep for a random time between 1 and 5 seconds. Finally, shut down the executor and handle proper termination using awaitTermination.
105. Write a program that uses an executor service to execute multiple Callable tasks. Each task should calculate and return the factorial of a number provided to it. Use Future objects to receive the results of the calculations. After all tasks are submitted, retrieve the results from the futures, print them, and ensure the executor service is shut down correctly.

Revision

1. Intro to Multi-threading
2. Creating a Thread
3. States of a Thread
4. Thread Priority
5. Join Method
6. Synchronize keyword
7. Thread Communication
8. Intro to Executor Service
9. Multiple Threads with Executor
10. Returning Futures

Practice Exercise

Multi threading & Executor Service

Answer in True/False

1. In Java, the main thread is the only user thread that is created automatically when a program starts.
2. The run() method is automatically called when a thread is started using the start() method.
3. A thread can be in both the Runnable and Running state at the same time.
4. Lower priority threads are executed first to ensure equal processing time for all threads.
5. The join() method causes the calling thread to immediately stop executing until the thread it joins with stops running.
6. Using the synchronized keyword can prevent two threads from executing a method simultaneously.
7. The notify() method wakes up all threads that are waiting on the object's monitor.
8. The ExecutorService must be explicitly shut down to terminate the threads it manages.
9. A Callable task cannot be submitted to an ExecutorService.
10. The Thread.yield() method forces a thread to stop its execution permanently.

Practice Exercise

Multi threading & Executor Service

Answer in True/False

1. In Java, the main thread is the only user thread that is created automatically when a program starts. — True
2. The run() method is automatically called when a thread is started using the start() method. — True
3. A thread can be in both the Runnable and Running state at the same time. — False
4. Lower priority threads are executed first to ensure equal processing time for all threads. — False
5. The join() method causes the calling thread to immediately stop executing until the thread it joins with stops running. — True
6. Using the synchronized keyword can prevent two threads from executing a method simultaneously. — True
7. The notify() method wakes up all threads that are waiting on the object's monitor. — False
8. The ExecutorService must be explicitly shut down to terminate the threads it manages. — True
9. A Callable task cannot be submitted to an ExecutorService. — False
10. The Thread.yield() method forces a thread to stop its execution permanently. — False

KG Coding

Some Other One shot Video Links:

- Complete HTML
- Complete CSS
- Complete JavaScript
- Complete React and Redux
- One shot University Exam Series

http://www.kgcoding.in/

Our YouTube Channels

KG Coding Android App

KG Coding

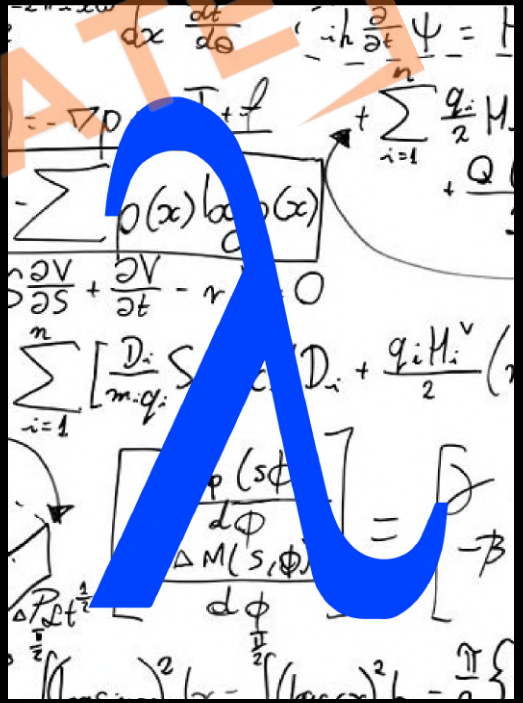
Knowledge GATE

KG Placement Prep

Sanchit Socket

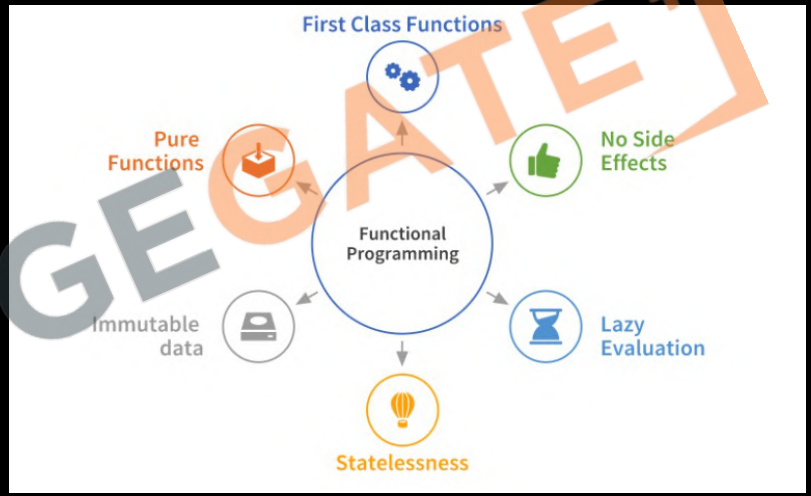
13 Functional Programming

1. What is Functional Programming
2. Lambda Expression
3. What is a Stream
4. Filtering & Reducing
5. Functional Interfaces
6. Method References
7. Functional vs Structural Programming
8. Optional Class
9. Intermediate vs Terminal Operations
10. Max, Min, Collect to List
11. Sort, Distinct, Map



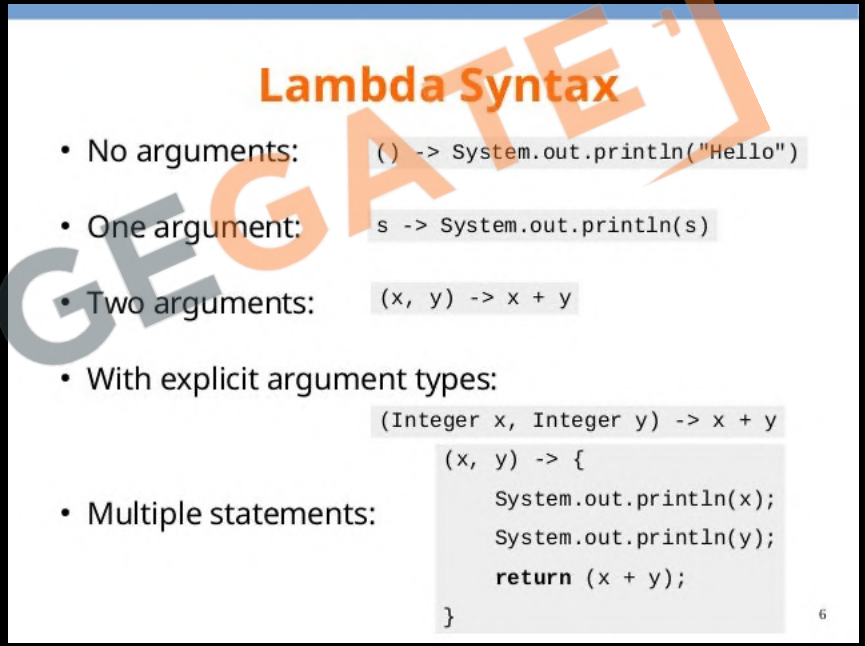
13.1 What is Functional Programming

1. Functional Programming: It's a way of writing programs where you use functions like small building blocks.
2. Functions as First-Class: Functions can be passed as arguments, returned from other functions, and assigned to variables.
3. Immutable Data: Once you create a piece of data, you don't change it.
4. Pure Functions: These are special functions that always give the same result for the same input with no Side-Effects.
5. Functional Interfaces: These are like templates for functions, making it easier to use them in different parts of your program.



13.2 Lambda Expression

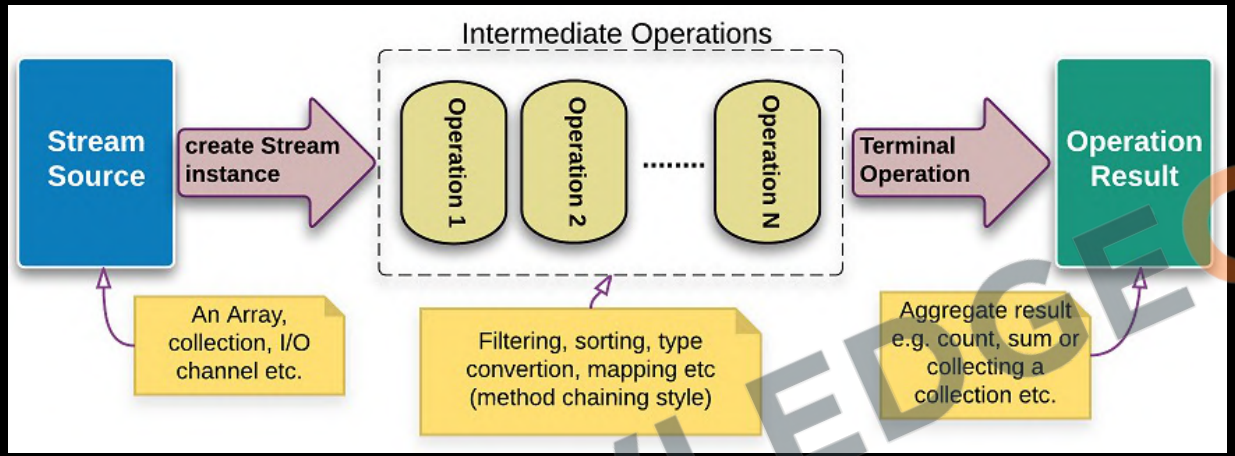
1. Shortcuts: Lambda expressions are quick, nameless functions for small tasks.
2. Syntax: Written as (parameters) -> {body}, linking inputs to actions.
3. Functional Interfaces: They work with interfaces that have only one method, making code concise.
4. Readability: They make code shorter and clearer, especially with collections.
5. Useful with Collections: Great for managing lists and sets, like filtering or sorting.



13.2 Lambda Expression

```
// Simple Addition:
(int a, int b) -> a + b;

// Check if a Number is Even:
(int number) -> number % 2 == 0;

// Print a Message:
(String message) -> System.out.println(message);

// Sort a List of Strings by Length:
(String s1, String s2) -> s1.length() - s2.length();

// Runnable with Lambda (No Parameters):
() -> System.out.println("Hello, World!");
```

13.3 What is a Stream



1. Element Sequence: Streams represent a sequence of elements
2. Functional Operations: Operations like map, filter, and reduce.
3. No Storage: Streams don't store data; they process it on-the-fly from sources like collections or arrays.
4. Efficiency: Stream operations can be lazy, processing elements only as needed, which is efficient for large data.
5. One-Time Use: Streams are consumable; once processed, they cannot be.
6. Parallel Capable: They support parallel processing, making operations faster by utilizing multiple threads.

13.4 Filtering & Reducing (Filter)

```
List<String> myList = List.of("apple", "banana", "cherry", "date");
myList.stream()
    .filter(s -> s.endsWith("e"))
    .forEach(s -> System.out.println(s));
```

1. Purpose: Used to filter elements of a stream based on a given predicate (a condition). Only elements that satisfy the condition are included in the resulting stream.
2. Lazy Operation: It's a lazy operation, meaning it's not executed until a terminal operation (like collect or forEach) is invoked on the stream.
3. Returns a Stream: filter itself returns a new stream with elements that match the predicate.

13.4 Filtering & Reducing (Reduce)

```
List<Integer> numbers = Arrays.asList(1, 2, 3, 4, 5);
int sum = numbers.stream()
                 .reduce(0, (a, b) -> a + b);
// sum: 15
```

1. Purpose: Used to reduce the elements of a stream to a single value. It takes a binary operator as a parameter and applies it repeatedly, combining the elements of the stream.
2. Versatile: Can be used for summing, finding min or max, and combining elements in a myriad of ways.
3. Optional or Default Value: Without an identity value, reduce returns an Optional. With an identity value, it returns a default value if the stream is empty.

CHALLENGE

106. Write a lambda expression that takes two integers and returns their multiplication. Then, apply this lambda to a pair of numbers.
107. Convert an array of strings into a stream. Then, use the stream to print each string to the console.
108. Given a list of strings, use stream operations to filter out strings that have length of 10 or more and then concatenate the remaining strings.
109. Given a list of integers, use stream operations to filter odd numbers and print them.

13.5 Functional Interfaces

1. Single Abstract Method (SAM): A functional interface has only one abstract method. However, it can have multiple default or static methods.
2. Lambda Compatibility: They are intended to be used with lambda expressions, providing a target type for lambdas and method references.
3. @FunctionalInterface Annotation: While not mandatory, this annotation helps the compiler to identify the intention of making an interface functional and to generate an error if the annotated interface does not satisfy the conditions.
4. Common Examples: Predicate, Consumer, BinaryOperator, Runnable, Callable, Comparator, and user-defined interfaces can be functional if they have only one abstract method.

```
Predicate<Integer> isPositive = x -> x > 0;
// Output: true
System.out.println(isPositive.test(5));
// Output: false
System.out.println(isPositive.test(-5));

Consumer<String> print =
        message -> System.out.println(message);
// Output: Hello, World!
print.accept("Hello, World!");

BinaryOperator<Integer> multiply = (a, b) -> a * b;
// Output: 15
System.out.println(multiply.apply(5, 3));
```

13.6 Method References

Lambda Expression	Method Reference
`s -> s.toLowerCase()`	`String::toLowerCase`
`s.toLowerCase()`	`String::toLowerCase`
`(a, b) -> a.compareTo(b)`	For Integers it will be `Integer::compareTo` and for strings it is `String::compareTo`
`(a, b) -> Person.compareByAge(a, b)`	`Person::compareByAge`

Syntax:

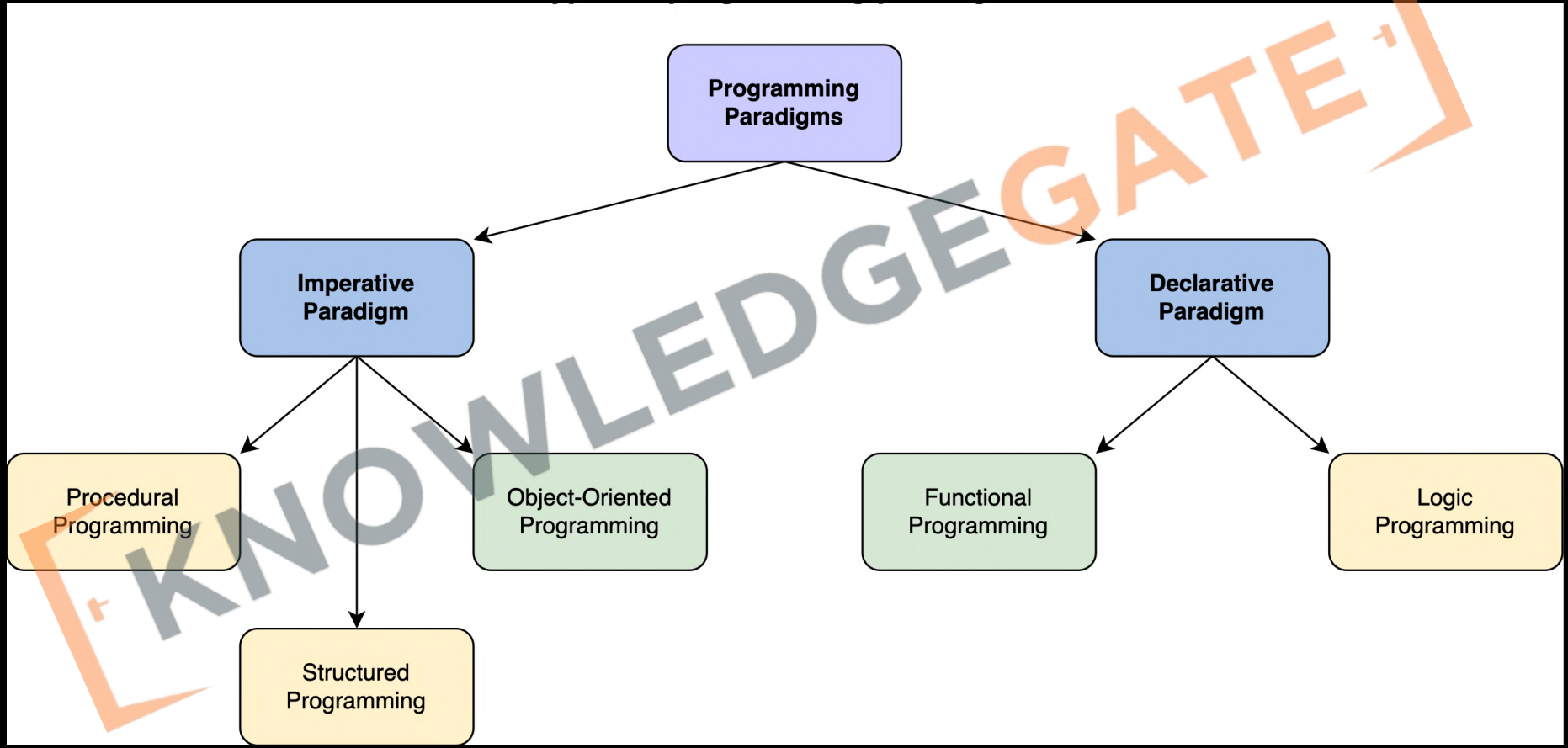
- Static Method References: ClassName::staticMethodName
- Instance Method: instance::instanceMethodName
- Instance Method Particular Class: ClassName::methodName
- Constructor References: ClassName::new.

1. Purpose Syntax & Usage: A method reference is described using :: (double colon) syntax. For example, System.out::println refers to the println method of the System.out object.
2. Functional Interfaces: They are used with functional interfaces.
3. Benefit: They make your code more readable and concise
4. Limitation: They can only be used for methods that fit the parameters and return type.

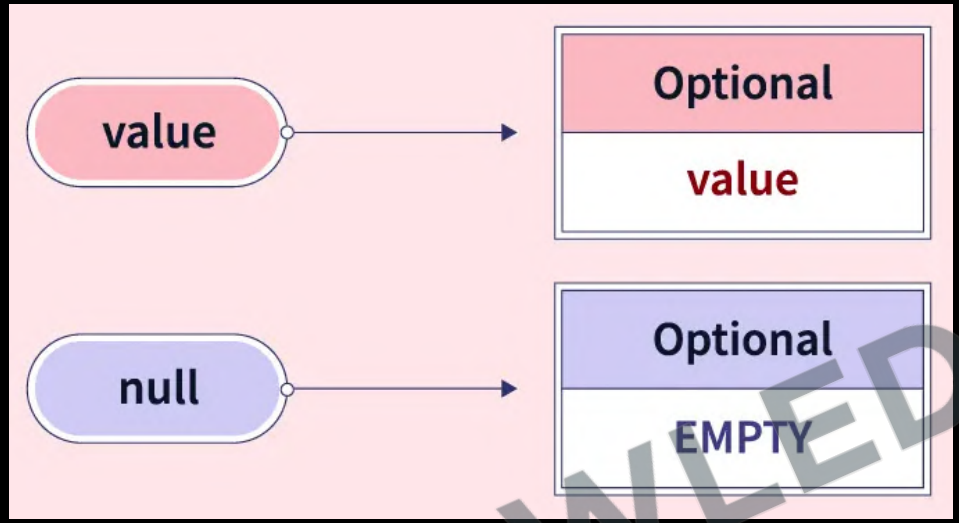
13.7 Functional vs Structural Programming

Imperative Programming	Declarative Programming
1. Computation	
You describe the step-by-step instructions for how an executed program achieves the desired results.	You set the conditions that trigger the program execution to produce the desired results.
2. Readability and complexity	
With the emphasis on the control flow, you can often follow the step-by-step process fairly easily. However, as you add more code, it can become longer and more complex	Step-by-step processes are eschewed. You'll discover that this paradigm is less complex and requires less code, making it easier to read.
3. Customization	
A straightforward way to customize and edit code and structure is offered. You have complete control and can easily adapt the structure of your program to your needs.	Customizing the source code is more difficult because of complicated syntax and the paradigm's dependence on implementing a pre-configured algorithm.
4. Optimization	
Adding extensions and making upgrades are supported, but doing so is significantly more challenging than with declarative programming, making it harder to optimize.	You can easily optimize code because an algorithm controls the implementation. Furthermore, you can add extensions and make upgrades.
5. Structure	
The code structure can be long and complex. The code itself specifies how it should run and in what order.	The code structure is concise and precise, and it lacks detail.

13.7 Functional vs Structural Programming



13.8 Optional Class



1. Creating Optional Objects: Optional.empty(), Optional.of(), Optional.ofNullable()
2. Checking Value Presence: isPresent() and ifPresent()
3. Default Values: orElse() and orElseGet()
4. Value Transformation: map()
5. Throwing Exception: orElseThrow()

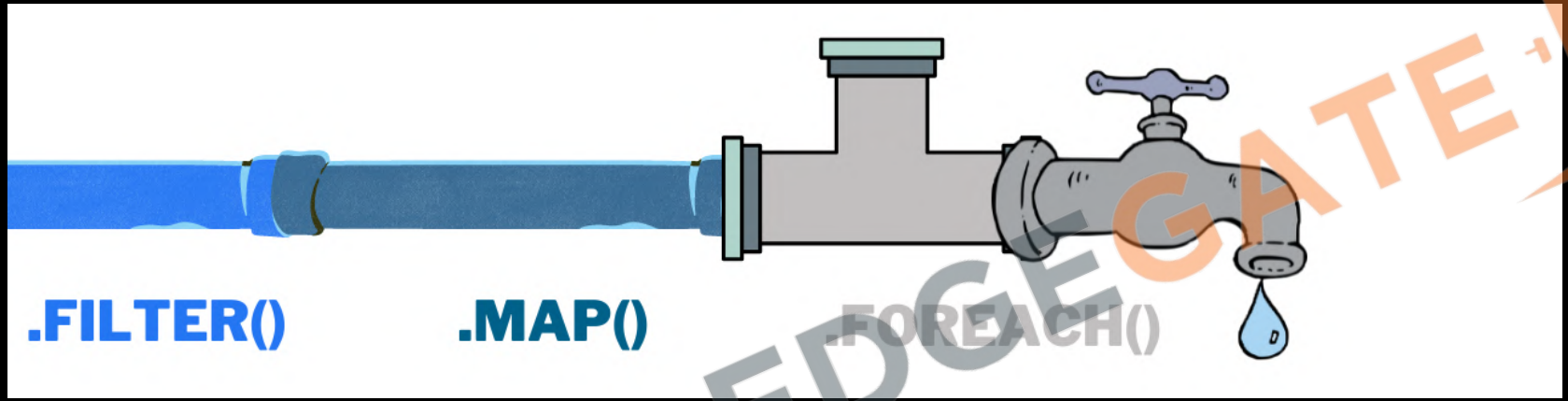
13.8 Optional Class

```java
// Creating Optional objects
Optional<String> optionalEmpty = Optional.empty();
Optional<String> optionalOf = Optional.of("Java");
Optional<String> optionalNullable = Optional.ofNullable(null);

// Checking presence of value
if (optionalOf.isPresent()) {
    System.out.println("Value is present: " + optionalOf.get());
}

// Using orElse to provide a fallback
String orElseExample = optionalEmpty.orElse("Default Value");
System.out.println("Using orElse: " + orElseExample);

// Using ifPresent to perform an action if value is present
optionalOf.ifPresent(System::out::println);
```

CHALLENGE

110. Create your own functional interface with a single abstract method that accepts an integer and returns a boolean. Implement it using a lambda that checks if the number is prime.
111. Write two versions of a program that calculates the factorial of a number: one using structural (procedural) programming, and the other using functional programming.
112. Write a function that accepts a string and returns an Optional<String>. If the string is empty or null, return an empty Optional, otherwise, return an Optional containing the uppercase version of the string.

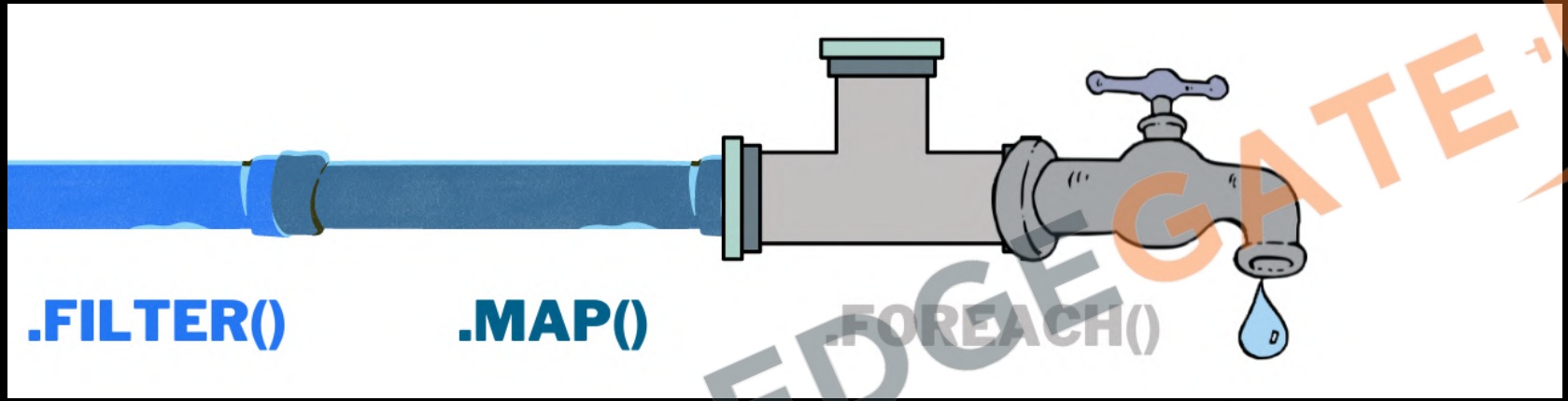
13.9 Intermediate vs Terminal Operations



Intermediate Operations

1. Laziness: Executed only when a terminal operation is invoked, setting up a pipeline without processing data.
2. Stream Transformation: Transform one stream into another, e.g., filter, map. They're chainable, allowing multiple transformations.
3. State Handling: Can be stateless (like map) or stateful(like sorted), affecting processing.

13.9 Intermediate vs Terminal Operations



Terminal Operations

1. Computation Trigger: Initiates the stream processing and closes the stream. After this, the stream can't be reused.
2. Final Outcome: Produces a result (like a sum or list) or a side-effect (like printing each element). Not chainable.
3. Examples: Operations like collect, forEach, reduce, sum, max, min, and count are terminal.

13.10 Max, Min, Collect to List

```java
List<Integer> numbers = List.of(4, 2, 5, 1, 3);
Optional<Integer> maxNumber = numbers.stream()
    .max(Integer::compareTo);
// Output: 5
maxNumber.ifPresent(System.out::println);

List<Integer> numbers = List.of(4, 2, 5, 1, 3);
Optional<Integer> minNumber = numbers.stream()
    .min(Integer::compareTo);
// Output: 1
minNumber.ifPresent(System.out::println);

List<String> words = Arrays.asList("Stream",
    "Operations", "Java");
List<String> collectedWords = words.stream()
    .collect(Collectors.toList());
// Output: [Stream, Operations, Java]
System.out.println(collectedWords);
```

1. max() finds the largest element in the stream according to a given comparator or natural ordering.
2. min() identifies the smallest element in the stream based on a provided comparator or natural ordering.
3. collect(Collectors.toList()) gathers all the elements of the stream into a new List.

13.11 Sort, Distinct, Map

```java
List<Integer> numbers = List.of(4, 2, 5, 1, 3);
List<Integer> sortedNumbers = numbers.stream()
    .sorted()
    .collect(Collectors.toList());
// Output: [1, 2, 3, 4, 5]
System.out.println(sortedNumbers);


List<String> items = List.of("apple",
    "banana", "apple", "orange", "banana");
List<String> distinctItems = items.stream()
    .distinct()
    .collect(Collectors.toList());
// Output: [apple, banana, orange]
System.out.println(distinctItems);

List<String> words = List.of("Stream",
    "Operations", "Java");
List<String> uppercaseWords = words.stream()
    .map(String::toUpperCase)
    .collect(Collectors.toList());
// Output: [STREAM, OPERATIONS, JAVA]
System.out.println(uppercaseWords);
```

1. sorted() orders the elements of a stream based on their natural order or a provided comparator.
2. distinct() filters out duplicate elements, ensuring that every element in the resulting stream is unique.
3. map() applies a function to each element of a stream, transforming them into a new stream of results based on the function logic.

CHALLENGE

113. Given an array of integers, create a stream, use the distinct operation to remove duplicates, and collect the result into a new list.
114. Create a list of employees with name and salary fields. Write a comparator that sorts the employees by salary. Then, use this comparator to sort your list using the sort stream operation.
115. Create a list of strings representing numbers ("1", "2", ...). Convert each string to an integer, then again calculating squares of each number using the map operation and sum up the resulting integers.

Revision

1. What is Functional Programming
2. Lambda Expression
3. What is a Stream
4. Filtering & Reducing
5. Functional Interfaces
6. Method References
7. Functional vs Structural Programming
8. Optional Class
9. Intermediate vs Terminal Operations
10. Max, Min, Collect to List
11. Sort, Distinct, Map

Practice Exercise

Functional Programming

Answer in True/False

1. Functions can't be assigned to variables, passed as arguments, or returned from other functions.
2. A lambda expression in Java can be used to implement any interface, regardless of the number of abstract methods.
3. A Java stream represents a sequence of elements and supports various methods which can be pipelined to produce the desired result.
4. The filter method in streams is a terminal operation that returns a boolean value.
5. A functional interface in Java is an interface with exactly one abstract method.
6. Method references in Java can only refer to static methods.
7. The Optional class in Java is used to avoid NullPointerException.
8. Intermediate operations on streams are executed immediately and are always followed by terminal operations.
9. The max and min operations on streams return an Optional describing the maximum or minimum element.
10. The sorted operation in streams is a terminal operation and it sorts the elements of the stream in their natural order.

Practice Exercise

Functional Programming

Answer in True/False

1. Functions can't be assigned to variables, passed as arguments, or returned from other functions. — False
2. A lambda expression in Java can be used to implement any interface, regardless of the number of abstract methods. — False
3. A Java stream represents a sequence of elements and supports various methods which can be pipelined to produce the desired result. — True
4. The filter method in streams is a terminal operation that returns a boolean value. — False
5. A functional interface in Java is an interface with exactly one abstract method. — True
6. Method references in Java can only refer to static methods. — False
7. The Optional class in Java is used to avoid NullPointerException. — True
8. Intermediate operations on streams are executed immediately and are always followed by terminal operations. — False
9. The max and min operations on streams return an Optional describing the maximum or minimum element. — True
10. The sorted operation in streams is a terminal operation and it sorts the elements of the stream in their natural order. — False

KG Coding

Some Other One shot Video Links:

- Complete HTML
- Complete CSS
- Complete JavaScript
- Complete React and Redux
- One shot University Exam Series

http://www.kgcoding.in/

Our YouTube Channels

KG Coding Android App

KG Coding

Knowledge GATE

KG Placement Prep

Sanchit Socket